Advances in Engineering Software 206 (2025) 103922

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Research paper

Integrating custom constitutive models into FEniCSx: A versatile approach and case studies

Sjard Mathis Rosenbusch *, Philipp Diercks, Vitaliy Kindrachuk, Jörg F. Unger

Bundesanstalt für Materialforschung und -prüfung (BAM), Unter den Eichen 87, 12205, Berlin, Germany

ARTICLE INFO

Keywords:
Finite element method
Constitutive models
FEniCSx
UMAT
Rust
Python
C++

ABSTRACT

The development and integration of user-defined constitutive relationships into finite element (FE) tools using standardized interfaces play a pivotal role in advancing the capabilities of FE solvers for structural mechanics applications. While commercial FE solvers like Abaqus and Ansys have designed their interfaces to provide custom stresses, tangents, and updated history variables, the open-source solver FEniCSx remains efficient only when the constitutive update has an analytical representation. This restricts the application of FEniCSx for non-linear structural mechanics.

Since FEniCSx has become a powerful and popular open-source tool for solving partial differential equations, particularly due to its automatic computation of Hessians, we aim to develop a generalized interface to enhance its capability for constitutive modeling. This approach will address complex constitutive equations that require iterative solutions at the quadrature point level. Specific implementation challenges, such as using return-mapping procedures, can then be managed commonly. The provided interface for custom constitutive models offers a versatile way to implement them in various languages, including C++, Python, Rust, and Fortran. Finally, with UMATs for viscoplastic models as an example, we demonstrate how existing user subroutines can be incorporated into the interface and utilized within the FEniCSx framework.

1. Introduction

Modeling and simulations are ubiquitous in many sectors of engineering and science. In particular, the understanding and prediction of materials behavior is an important factor in the development of new materials, products and applications. Commercial finite element (FE) codes such as Abaqus, Ansys and LS-DYNA, just to mention a few, offer many features for the modeling of structural problems, including interfaces for the implementation of custom elements, constitutive laws etc. These commercial codes are robust and high-performance solvers, but users may face difficulties if their use case is not covered and cannot be addressed through the available subroutines. For instance, the number of partial differential equations (PDEs) that can be solved with the commercial tool may be limited, and in general it is not possible to make changes to the code base. In this case, open source FE codes are an attractive alternative due to their flexibility. Typically these codes are designed for research purposes, allowing the solution of any PDE and providing the possibility to alter the code base to meet the specific requirements of a particular problem. One prominent example is the FEniCS computing platform. Throughout the article we consider the next generation of the FEniCS computing platform (FEniCSx), although the same ideas can be applied to the legacy version of FEniCS [1]. It comprises the following components:

- the Unified Form Language (UFL) [2] for definition of variational forms,
- the FEniCSx Form Compiler[1] (FFCX) to generate low-level code from UFL forms,
- a library for FE basis evaluation (Basix [3,4]) and
- the FE library and problem-solving environment DOLFINx [5], which is a re-write of DOLFIN[2] [6].

For a more detailed description we refer to the associated publications.

In contrast to other open-source FE software that support structural mechanics and offer functionality to implement custom constitutive laws – such as CalculiX [7], deal.ii [8], Code_Aster [9], MOOSE [10] combined with external material libraries like NEML2, or Elmer [11]

* Corresponding author.
E-mail addresses: sjard-mathis.rosenbusch@bam.de (S.M. Rosenbusch), philipp.diercks@bam.de (P. Diercks), vitaliy.kindrachuk@bam.de (V. Kindrachuk), joerg.unger@bam.de (J.F. Unger).

[1] https://github.com/FEniCS/ffcx

[2] Dynamic Object-oriented Library for FINite element computation (DOLFIN).

https://doi.org/10.1016/j.advengsoft.2025.103922
Received 23 October 2024; Received in revised form 17 February 2025; Accepted 30 March 2025
Available online 20 April 2025

interfacing Abaqus UMATs [12] – the UFL still requires further development in this regard. It requires an explicit formulation of the weak form in terms of the unknown solution field. That is, the stress tensor has to be explicitly defined as a function of the displacement field, or more precisely, as a function of its gradient. However, in many constitutive relations, the stress depends implicitly on the displacement and is computed from history variables. This relation is often nonlinear and the solution of the constitutive law requires operations on the integration point level. Thus, the implementation of complex material models within the FEniCSx computing platform requires special treatment, and this paper presents one possible solution as its primary contribution. This solution has been previously implemented as a Python package `fenics-constitutive` by this paper's authors and others [13]. Therefore, this paper introduces `fenics-constitutive` and its capabilities.

The main idea of the approach used here is to fill the FE coefficient vectors that hold the values of the stress and stiffness tensors at each integration point by means of *any* external program or code. This idea was first presented in [14]. Herein, the authors use MFront [15], a code generation tool to handle material behavior (i. e. solve the constitutive law) and to fill the FE coefficient vectors. This approach was later incorporated into the Python packages `dolfinx_materials` [16] and `fenics-constitutive` which were developed independently. At the time of writing, `dolfinx_materials` supports a wider range of constitutive models within a single interface, such as hyperelasticity using the Hencky strain measure with automated generation of weak forms. In contrast, `fenics-constitutive` focuses on a simpler interface for small strain constitutive models, flexibility in assigning specific constitutive models to different parts of the FE model, and versatility. Namely, the fixed interface allows users to share code for their constitutive models and integrate them into their own projects with minimal effort. Note further that a recent contribution [17] employed constitutive models written in Rust together with FEniCSx for gradient-enhanced plasticity in explicit dynamics.

Another framework for expressing general constitutive models in FEniCSx [18–20] is based on the concept of external operators in UFL [21]. External operators allow the variational form to contain symbolic objects that are not inherently part of the UFL. The user is required to define the behavior of these objects (representing stress and its derivatives), for which the authors rely on pure Python functions, Numba [22] for just-in-time compilation (JIT) and JAX [23] for automatic differentiation (AD). Their approach provides a new technical possibility for including complex constitutive models within the FEniCSx framework, however to our knowledge there exists no software project that combines a prescribed constitutive model interface with external operators to facilitate interchangeability of models.

We, therefore, present the idea of the general interface for material models formulated for small strains in `fenics-constitutive`. The concept enables to integrate *any* program (written in any programming language, as long as the code can be linked to Python) that defines the relationship between stress and strain into FEniCSx. This interface represents our primary contribution, offering a versatile solution, initially as a proof of concept. As a general challenge regarding the implementation of constitutive models, we mention the computation of the algorithmically consistent tangent. With the approach suggested in this paper, the user can keep full control over the code that is used to define the material behavior, but also use the latest AD technology if desired. Moreover, as the external program is only required to return stress and its derivative given a strain state, the integration of solvers for boundary value problems defined on the micro-scale (FE^2), projection-based reduced order models, or neural networks is straightforward.

Building on the generic and consistent interface, our second contribution is to advance the tool-independent implementation of constitutive models that can be utilized with nearly any FE code in the future. In particular, we envision constitutive models developed in conjunction with said interface to be open source, tested, and re-usable by others. Recognizing the significant utility of the interfaces designed for the commercial FE tools, we present in addition their utilization with the FE software FEniCSx on the example of Abaqus UMAT subroutines. The interface offers users the opportunity to seamlessly integrate their existing routines with FEniCSx, thereby capitalizing on the extensive capabilities of both platforms. Additionally, users can benefit from the inherent flexibility and openness of the FEniCSx framework, enabling them to take advantage of an open-source environment for their computational modeling endeavors.

The rest of the paper is organized as follows. In Section 2 the class of mechanics problems considered in this paper, and how these are defined in FEniCSx is described. Furthermore, details on the available data structures and design choices for the FE interface for constitutive models are given. A detailed technical description of the Python interface and instructions for the compilation and binding of external programs written in C++, Rust and UMATs is given in Section 3. The numerical examples shown in Section 4 comprise a linear elastic problem to showcase the implementation with different programming languages and two benchmark problems utilizing an Abaqus UMAT in which a direct comparison between Abaqus and FEniCSx is made. Conclusions regarding the findings presented in this paper are given in Section 5.

1.1. Notation

Throughout the paper a direct tensor notation is used where applicable. Vectors are denoted by lowercase bold italic letters and second-order tensors by uppercase bold italic or Greek letters. Fourth-order tensors are symbolized like $\mathbb{C}$. A dot $\cdot$ represents a scalar contraction. If more than one contraction is carried out, the number of dots corresponds to the number of contractions, e.g.,

$$(\boldsymbol{a} \otimes \boldsymbol{b} \otimes \boldsymbol{c}) \cdot\cdot (\boldsymbol{d} \otimes \boldsymbol{e}) = \boldsymbol{a}(\boldsymbol{b} \cdot \boldsymbol{d})(\boldsymbol{c} \cdot \boldsymbol{e}) \,.$$

The scalar product between second-order tensors is thus

$$\boldsymbol{\sigma} \cdot\cdot \boldsymbol{\varepsilon} = \sigma_{ij} \boldsymbol{e}_i \otimes \boldsymbol{e}_j \cdot\cdot \varepsilon_{kl} \boldsymbol{e}_k \otimes \boldsymbol{e}_l = \sigma_{ij} \varepsilon_{ij} \,.$$

The nabla operator is introduced as

$$\boldsymbol{\nabla} := \frac{\partial}{\partial x_i} \boldsymbol{e}_i \,,$$

and the gradient of a vector field $\boldsymbol{v}$ is denoted by

$$\boldsymbol{v} \otimes \boldsymbol{\nabla} = \frac{\partial v_i}{\partial x_j} \boldsymbol{e}_i \otimes \boldsymbol{e}_j \,.$$

For further details on the tensor notation used in this manuscript, the reader is referred to Bertram and Glüge [24].

2. Mechanics problem in FEniCSx

The class of problems considered in this paper are small strain models (geometrically linear theory) with material nonlinearities. The latter comprises the Cauchy stress $\boldsymbol{\sigma} = f(\boldsymbol{\varepsilon}, \boldsymbol{Z})$ given as a nonlinear function of history (internal) variables $\boldsymbol{Z}$, which can be scalars or tensors (denoted as 2nd-order tensors), and the linear strain tensor

$$\boldsymbol{\varepsilon}(\boldsymbol{u}) := \frac{1}{2} (\boldsymbol{u} \otimes \boldsymbol{\nabla} + \boldsymbol{\nabla} \otimes \boldsymbol{u}) \,, \tag{1}$$

where $\boldsymbol{u}$ is the displacement field.

2.1. Weak form with residual and tangent

The governing equation for problems considered in this paper is the balance of linear momentum in the static case

$$\boldsymbol{\sigma} \cdot \boldsymbol{\nabla} = \rho(\boldsymbol{a} - \boldsymbol{b}) \,, \tag{2}$$

with body forces $\boldsymbol{b}$ and acceleration, which is assumed to vanish, $\boldsymbol{a} = \boldsymbol{0}$, without loss of generality. Eq. (2) must be satisfied for any material point within the continuum body, $\forall \boldsymbol{x} \in \Omega$, and in addition the following *boundary conditions* need to be defined for points on the boundary $\partial\Omega$:

$$\boldsymbol{u}(\boldsymbol{x}) = \hat{\boldsymbol{u}}(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \Gamma_{\mathrm{D}}, \tag{3}$$

$$\boldsymbol{\sigma}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) = \hat{\boldsymbol{t}}(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \Gamma_{\mathrm{N}}. \tag{4}$$

Here, Eq. (3) denotes a Dirichlet boundary condition with prescribed displacement $\hat{\boldsymbol{u}}$ on the boundary Γ_{D} and Eq. (4) denotes a Neumann boundary condition with prescribed traction vector $\hat{\boldsymbol{t}}$ on the boundary Γ_{N}. The weak form for the FE method is derived in the usual way (see e.g. [25]) by multiplication with a test function $\delta\boldsymbol{u}$, integration by parts, and application of Gauss's theorem:

$$\int_{\Omega} \delta\boldsymbol{u} \otimes \boldsymbol{\nabla} \cdot\cdot\, \boldsymbol{\sigma} \,\mathrm{d}\boldsymbol{x} = \int_{\Gamma_{\mathrm{N}}} \hat{\boldsymbol{t}} \cdot \delta\boldsymbol{u} \,\mathrm{d}s + \int_{\Omega} \rho\boldsymbol{b} \cdot \delta\boldsymbol{u} \,\mathrm{d}\boldsymbol{x}\,. \tag{5}$$

Note that

$$\delta\boldsymbol{u} \otimes \boldsymbol{\nabla} \cdot\cdot\, \boldsymbol{\sigma} = \frac{1}{2}\left(\delta\boldsymbol{u} \otimes \boldsymbol{\nabla} + \boldsymbol{\nabla} \otimes \delta\boldsymbol{u}\right) \cdot\cdot\, \boldsymbol{\sigma}\,, \tag{6}$$

because $\boldsymbol{\sigma}$ is symmetric. Using Eq. (1), Eq. (5) is rewritten as

$$\int_{\Omega} \boldsymbol{\varepsilon}(\delta\boldsymbol{u}) \cdot\cdot\, \boldsymbol{\sigma} \,\mathrm{d}\boldsymbol{x} = \int_{\Gamma_{\mathrm{N}}} \hat{\boldsymbol{t}} \cdot \delta\boldsymbol{u} \,\mathrm{d}s + \int_{\Omega} \rho\boldsymbol{b} \cdot \delta\boldsymbol{u} \,\mathrm{d}\boldsymbol{x}\,. \tag{7}$$

Finally, the weak form is linearized using a Taylor series expansion, neglecting higher order terms, which yields

$$\int_{\Omega} \boldsymbol{\varepsilon}(\delta\boldsymbol{u}) \cdot\cdot\, \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} \cdot\cdot\, \boldsymbol{\varepsilon}(\boldsymbol{u}) \,\mathrm{d}\boldsymbol{x} = \int_{\Gamma_{\mathrm{N}}} \hat{\boldsymbol{t}} \cdot \delta\boldsymbol{u} \,\mathrm{d}s + \int_{\Omega} \rho\boldsymbol{b} \cdot \delta\boldsymbol{u} \,\mathrm{d}\boldsymbol{x} - \int_{\Omega} \boldsymbol{\sigma} \cdot\cdot\, \boldsymbol{\varepsilon}(\delta\boldsymbol{u}) \,\mathrm{d}\boldsymbol{x}\,, \tag{8}$$

with the displacement field $\boldsymbol{u}$ as the sought after solution field. This weak form can be expressed in the UFL if the quantities $\frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}}$ and $\boldsymbol{\sigma}$ are given as functions of the *Quadrature* family[3] to be computed in an external program. In terms of the UFL, the right hand side is also denoted as the residual

$$R(\delta\boldsymbol{u}) := \int_{\Gamma_{\mathrm{N}}} \hat{\boldsymbol{t}} \cdot \delta\boldsymbol{u} \,\mathrm{d}s + \int_{\Omega} \rho\boldsymbol{b} \cdot \delta\boldsymbol{u} \,\mathrm{d}\boldsymbol{x} - \int_{\Omega} \boldsymbol{\sigma} \cdot\cdot\, \boldsymbol{\varepsilon}(\delta\boldsymbol{u}) \,\mathrm{d}\boldsymbol{x}\,, \tag{9}$$

with the left hand side as its Jacobian

$$J(\boldsymbol{v}, \delta\boldsymbol{u}) := \int_{\Omega} \boldsymbol{\varepsilon}(\delta\boldsymbol{u}) \cdot\cdot\, \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} \cdot\cdot\, \boldsymbol{\varepsilon}(\boldsymbol{v}) \,\mathrm{d}\boldsymbol{x}\,. \tag{10}$$

2.2. Design of FE interface for constitutive models

The linear form Eq. (9) and the bilinear form Eq. (10) are assembled into a residual vector $\boldsymbol{r}$ and a stiffness matrix $\boldsymbol{K}$. The solution of the nonlinear problem is then iteratively determined by the Newton–Raphson algorithm

$$\boldsymbol{u}_{k+1}^{n+1} = \boldsymbol{u}_{k}^{n+1} - \boldsymbol{K}^{-1}(\boldsymbol{u}_{k}^{n+1})\boldsymbol{r}(\boldsymbol{u}_{k}^{n+1}), \quad \boldsymbol{u}_{0}^{n+1} = \boldsymbol{u}^{n}, \tag{11}$$

where k denotes the iteration number. In essence, the task of the constitutive model at this point is to provide the stresses $\boldsymbol{\sigma}$ and the tangents $\partial\boldsymbol{\sigma}/\partial\boldsymbol{\varepsilon}$ at the new time increment $n+1$ based on the previous displacements $\boldsymbol{u}^n$, and thus strains, and internal state variables of the material (e. g., plastic strain in a plasticity model) in order to evaluate $\boldsymbol{r}$ and $\boldsymbol{K}$ in Eq. (11).

For small strains, a constitutive model can be either formulated in terms of a total strain, or in terms of strain increments. An incremental formulation is needed when large deformations with an objective stress rate are considered [26]. At time of writing, `fenics-constitutive` does not have an implementation of an objective stress rate. However, this is a goal considered in the design of the general interface for the constitutive models. Therefore an incremental formulation is considered. If a user wants to describe their model in terms of a total infinitesimal strain, they can treat the total strain as an additional history variable. Using the gradient of the displacement increment $\boldsymbol{\delta}^n = \boldsymbol{u}^{n+1} - \boldsymbol{u}^n$

$$\boldsymbol{\delta}^n \otimes \boldsymbol{\nabla} = \frac{\partial\boldsymbol{\delta}^n}{\partial\boldsymbol{x}^{n+1/2}} \tag{12}$$

at the midpoint configuration $\boldsymbol{x}^{n+1/2} = 0.5(\boldsymbol{x}^{n+1} + \boldsymbol{x}^n)$ directly allows the use of the Jaumann stress rate according to [26] since the strain increment

$$\Delta\boldsymbol{\varepsilon}^n = \frac{\boldsymbol{\delta}^n \otimes \boldsymbol{\nabla} + \boldsymbol{\nabla} \otimes \boldsymbol{\delta}^n}{2}, \tag{13}$$

and the spin tensor

$$\boldsymbol{W}^n := \frac{\boldsymbol{\delta}^n \otimes \boldsymbol{\nabla} - \boldsymbol{\nabla} \otimes \boldsymbol{\delta}^n}{2} \tag{14}$$

are directly determined from $\boldsymbol{\delta}^n \otimes \boldsymbol{\nabla}$. Therefore, we propose to use $\boldsymbol{\delta}^n \otimes \boldsymbol{\nabla}$ as input for the constitutive models. Note that for small strain models, it is assumed that no change in configuration takes place and all calculations are performed on the initial configuration $\boldsymbol{x}^0$. For the future extension of `fenics-constitutive` to large deformations with objective stress rates, an updated Lagrangian formulation with mesh update will be implemented. The feasibility of such an implementation has been demonstrated in [17] where an updated Lagrangian formulation for explicit dynamics has been implemented in FEniCSx. This is achieved by adding the increment in displacements to the mesh nodes after each load step.

Furthermore, rate-dependent material models like viscoelastic or viscoplastic models require information about the time which can be provided by the time at the beginning of the load step t^n and the time increment Δt^n. The constitutive model can be considered as a function Σ that updates the stress $\boldsymbol{\sigma}^{n+1}$, the internal variables $\boldsymbol{Z}^{n+1}$ and computes the tangent $\mathbb{C}^{n+1} = \partial\boldsymbol{\sigma}/\partial\boldsymbol{\varepsilon}$

$$(\boldsymbol{\sigma}^{n+1}, \mathbb{C}^{n+1}, \boldsymbol{Z}^{n+1}) = \Sigma(\boldsymbol{\sigma}^n, \boldsymbol{\delta}^n \otimes \boldsymbol{\nabla}, \boldsymbol{Z}^n, t^n, \Delta t^n). \tag{15}$$

from the gradient of the increment in displacements $\boldsymbol{\delta}^n \otimes \boldsymbol{\nabla}$, times t^n and Δt^n and the previous values $\boldsymbol{\sigma}^n$ and $\boldsymbol{Z}^n$.

2.3. Manipulation of quadrature point data

In the UFL *Quadrature* elements represent finite elements with degrees of freedom being function evaluations at quadrature, i. e. integration points. The definition of such a quadrature element comprises at least the degree of the polynomial that is to be integrated exactly, determining the number of quadrature points per cell, and the value shape of the elements of the associated quadrature space Q. The latter determines the number of values that need to be stored at each quadrature point and corresponds e. g. to the number of independent components of the Cauchy stress tensor $\boldsymbol{\sigma}$ – i.e. 6 if stored as a vector in Voigt notation or 9 if stored as a 3×3 tensor. The dimension of a quadrature space $\dim(Q)$ is the total number of quadrature points in the computational grid times the number of values that are stored at each quadrature point.

Since the quadrature element represents only the function evaluation at the quadrature points and no shape functions are defined for this finite element, the sole purpose of a function $f \in Q$ is to store data at the quadrature points. For example, using the class `dolfinx.fem.Expression`[4] the strain can be computed by evaluating a UFL expression (i.e. Eq. (1)) at the quadrature points in all cells of the computational grid. Moreover, there is no coupling between elements and thus the data is contiguously stored in memory. Therefore, the data can be accessed easily via e.g. a **for loop**. Moreover, the data-oriented

[3] https://docs.fenicsproject.org/ufl/2024.1.0/manual/examples.html#the-quadrature-family

[4] https://docs.fenicsproject.org/dolfinx/main/python/generated/dolfinx.fem.html#dolfinx.fem.Expression

and functional design of DOLFINx allows data to be passed between languages (see [5], sec. 2). Thus, in Python, the data, that is the entries of the FE coefficient vectors, is wrapped by a `numpy.ndarray` and directly accessible through the Python interface. This means, regarding the finite element formulation derived from the balance of linear momentum (see Section 2.1), and integrating constitutive models beyond the UFL, the main task for a material routine used in conjunction with FEniCSx is to *update* the FE coefficient vectors for the Cauchy stress $\boldsymbol{\sigma}$ and its derivative $\partial\boldsymbol{\sigma}/\partial\boldsymbol{\epsilon}$ in accordance to the current deformation state. In the simplest case, this can be a pure Python function acting on NumPy arrays, as shown in Section 4.1.

2.4. Nonlinear problems in FEniCSx

The module `dolfinx.fem` provides methods to discretize FE matrices and vectors, based on bilinear and linear forms, respectively, expressed in the UFL. At the time of writing, as linear algebra backend, either the built-in module `dolfinx.la` or optionally PETSc [27] can be used. For discretizations based on PETSc the class `dolfinx.fem.petsc.NonlinearProblem` can be used to define nonlinear problems. As input to this class, the weak form of the PDE in residual form Eq. (9), and optionally its Jacobian Eq. (10) are required. Moreover, an implementation of a Newton solver is provided by `dolfinx.nls.petsc.NewtonSolver` which can be used together with `dolfinx.fem.petsc.NonlinearProblem`.

With respect to the implementation of the mechanics problems considered, this means that given implementations of Eq. (9) and Eq. (10), we can implement our `IncrSmallStrainProblem` by inheriting from `dolfinx.fem.petsc.NonlinearProblem` to be able to use `dolfinx.nls.petsc.NewtonSolver`. We then need to make sure that in each iteration the values for the stress $\boldsymbol{\sigma}$ and the algorithmic tangent $\partial\boldsymbol{\sigma}/\partial\boldsymbol{\epsilon}$ (and any history variables) are correctly updated before the discrete system is assembled and solved, which can be achieved, for example, by appropriately overriding the method `dolfinx.fem.petsc.NonlinearProblem.form`.

3. Abstract Python interface and compilation of user-defined constitutive models

The previously defined constitutive model structure has been implemented in `fenics-constitutive` [13] as an abstract base class named `IncrSmallStrainModel` (see listing 1). This class serves as the foundation for defining custom constitutive models by requiring the implementation of specific methods.

```python
class IncrSmallStrainModel(ABC):

  @abstractmethod
  def evaluate(
    self,
    t: float,
    del_t: float,
    grad_del_u: np.ndarray,
    stress: np.ndarray,
    tangent: np.ndarray,
    history: dict[str, np.ndarray] | None,
  ) -> None:

  @abstractproperty
  def constraint(self) -> StressStrainConstraint:

  @property
  def stress_strain_dim(self) -> int:
    return self.constraint.stress_strain_dim

  @property
  def geometric_dim(self) -> int:
    return self.constraint.geometric_dim

  @abstractproperty
  def history_dim(self) ->
    dict[str, int | tuple[int, int]] | None:
```

Listing 1: Abstract base class for a constitutive model in `fenics-constitutive`.

The interface includes the `evaluate` method, which accepts `numpy.ndarray` arrays to update stress states, tangents, and history variables in-place. The method `history_dim` specifies the number of history variables to be created by the solver. These arrays represent data across all quadrature points, and the implementation is responsible for efficiently handling this looping internally. Additionally, the methods `constraint`, `stress_strain_dim`, and `geometric_dim` provide information about the constraint applied to the stresses and strains (e.g., uniaxial strain or stress, plane strain), stress/strain vector dimensions, and geometric dimensionality.

The implementation uses Mandel notation [28] for stresses and strains – a reduced representation for symmetric tensors with 6 components instead of 9. Both Mandel and the often used Voigt notations are suitable for constitutive model implementations, but Mandel notation offers advantages, such as using a factor of $\sqrt{2}$ for shear components instead of 2. This ensures stress and strain share the same basis, and the inner products of stress or strain in Mandel notation match the full tensor inner products. The order of stress and strain components can be chosen arbitrarily, however, we follow the order used in Abaqus Implicit:

$$\begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix} \mapsto \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sqrt{2}\sigma_{12} \\ \sqrt{2}\sigma_{13} \\ \sqrt{2}\sigma_{23} \end{bmatrix}. \tag{16}$$

Given that CPython – the default Python interpreter – is implemented in C, external libraries in other languages can interact seamlessly by adhering to Python data structures defined in C. One goal is to compile the implementation into shared libraries (e.g., `.so` files on Linux or `.dll` files on Windows) interpretable by CPython.

The `numpy.ndarray` data type is widely supported by libraries that offer Python bindings, enabling in-place data manipulation using extensions written in C/C++, Rust, and other high-performance languages. Furthermore, since `dolfinx.fem.Function` objects expose their underlying data as `numpy.ndarray`, Python can efficiently connect the fast FEniCSx libraries with custom constitutive models written in other languages.

In the following subsections, we will briefly demonstrate how to implement, compile and import constitutive models written in Rust, C++ and Fortran (as UMATs). These approaches have been used for the examples of linear elasticity from Section 4.1. Since the datatype `StressStrainConstraint` is not available in C++ or Rust, the implemented models follow a simpler interface and will need to be wrapped in another class in the Python code in order to satisfy the interface.

The full code for the following examples is published on Zenodo [29].

3.1. Pure C++/Cmake project

For writing C++ code, we recommend using `CMake` to manage compilation and linking. For Python bindings, the `pybind11` or `nanobind` [30] libraries can be used. The latter is employed by FEniCSx for its Python bindings and both support datatypes from `Eigen` [31], a linear algebra library suitable for small matrices and

vectors — essential for constitutive model implementations. Additionally, CMake and a C++ compiler (such as Clang or GCC) need to be installed. Dependencies can be system-wide or managed within an isolated environment like conda.

The model is compiled using cmake, generating a shared object file named elasticity_cpp.cpython-310-x86_64-linux-gnu.so, which can be imported in Python as import elasticity_cpp when in the current working directory. For comparability between Rust and C++ results, the Clang compiler is recommended due to its shared backend with Rust. However, GCC can also be used.

Within main.cpp, a class Elasticity3D and its binding code are defined. While implementation details are omitted here, users mainly need to implement their models within the for-loop in listing 2. Using Eigen::Ref allows pybind11 to access numpy.ndarray data without copies – critical for the in-place modifications required by the design from listing 1.

```cpp
struct Elasticity3D
{
  Eigen::Matrix<double, 6, 6> D_;
  Elasticity3D(double E, double nu)
  {
    //create elasticity tensor D_ from parameters
  }
  void evaluate(
    double t,
    double del_t,
    const Eigen::Ref<const Eigen::VectorXd> &
    del_grad_u,
    Eigen::Ref<Eigen::VectorXd> &stress,
    Eigen::Ref<Eigen::VectorXd> &tangent,
    std::map<std::string, Eigen::Ref<Eigen::
    VectorXd>> &history)
  {
    int n_ip = del_grad_u.size() / 9;
    for (int ip = 0; ip < n_ip; ip++)
    {
      //code for matrix-vector product D_ *
    del_strain
      //and writing D_ into &tangent
    }
  }
  std::map<std::string, int> history_dim() {
    return {};
  }
};
```

Listing 2: Example of C++ constitutive model class.

3.2. Pure Rust/Cargo project

Rust projects are similar to C++ projects, but use Cargo instead of CMake for compiling and linking. The Cargo.toml file defines build instructions and dependencies.

For linear algebra with small matrices and vectors, the nalgebra library is commonly used, while the PyO3 library facilitates creating Python modules. The numpy package supports direct handling of numpy.ndarray as native Rust types, particularly for vectors and matrices from nalgebra.

The binary is built with cargo build --release. Unlike the C++ project, the shared object file needs to be renamed manually or linked under Linux as elasticity_rs.so to be Python-compatible.

```rust
#[pyclass]
struct Elasticity3D {
  D: SMatrix<f64, 6, 6>,
}
#[pymethods]
impl Elasticity3D {
  #[new]
  fn new(E: f64, nu: f64) -> PyResult<Self> {
    //create elasticity tensor D
  }
  fn evaluate(
    &self,
    t: f64,
    del_t: f64,
    del_grad_u: PyReadonlyArray1<f64>,
    stress: PyReadwriteArray1<f64>,
    tangent: PyReadwriteArray1<f64>,
    history: HashMap<String, PyReadwriteArray1<f64
    >>,
  ) -> PyResult<()> {
    //convert numpy arrays to nalgebra matrices
    for ip in 0..n_ip {
      //evaluate matrix-vector-product self.D *
    del_strain
      //and write self.D to tangent
    }
    Ok(())
  }
  fn history_dim(&self) -> HashMap<String, i32> {
    HashMap::new()
  }
}
```

Listing 3: Example of Rust constitutive model class.

In this setup, a struct encapsulates the model parameters (e.g., the elasticity tensor), and methods are implemented within an impl block (see listing 3). The macros #[pyclass] and #[pymethods] from PyO3 enable Python to recognize the Rust struct as a Python class. Special types like PyReadonlyArray1 and PyReadwriteArray1 handle immutable and mutable numpy.ndarray respectively. Converting these to native Rust types (like nalgebra matrices) is essential for performing numerical operations.

3.3. Binding code for UMAT

The binding of a UMAT to Python is enabled by a mixed-language approach. Note that UMATs may be written in any programming language that supports the C ABI, such as Fortran, C/C++ and Rust. However, Fortran is the default language when writing UMATs and is therefore the focus of this section. The UMAT's Fortran code is compiled into a shared object file, which is then dynamically linked to a C++ code that handles the integration point loop and links it to Python via pybind11. Therefore, the layout of the code is similar to that of the C++ project, including the CMakeLists.txt. The C++ constitutive model class is called Umat3D and takes the path to the compiled UMAT as input. The obtained shared object is called umat.cpython-310-x86_64-linux-gnu.so.

Since the UMAT expects the stresses and strains in Voigt notation, some conversion steps from and to Mandel notation are required. However, for complex material models, the additional computing time is negligible compared to the evaluation of the material model itself. If we consider the tangent $\mathbb{C} \in \mathbb{R}^{6\times6}$ that is returned by the UMAT to be a block matrix with $\mathbb{C}_{ij} \in \mathbb{R}^{3\times3}$

$$\begin{bmatrix} \mathbb{C}_{11} & \mathbb{C}_{12} \\ \mathbb{C}_{21} & \mathbb{C}_{22} \end{bmatrix}, \tag{17}$$

it needs to be transformed to

$$\begin{bmatrix} \mathbb{C}_{11} & \sqrt{2}\mathbb{C}_{12} \\ \sqrt{2}\mathbb{C}_{21} & 2\mathbb{C}_{22} \end{bmatrix}, \tag{18}$$

in order to be compatible with vectors in Mandel notation. The conversion between Voigt notation and Mandel notation is done by changing the factors of the shear components. Note that a subset of UMAT arguments required for integrating small-strain models is usable in `fenics-constitutive`. These include strain and its increment, stress and its algorithmic tangent, history variables, and time increment. The interaction occurs via the `Umat3D` class.

In order to improve performance, the `Umat3D` class is designed with all dimensions (stress, geometrical, history) as constants that are known at compile-time. Templates are used to make the class generic for the history dimension and in all examples in this paper, the class is written for the 3D case only. This means that for each possible size of history dimensions that are investigated in this paper, a new class has to be exported by `pybind11`, meaning `Umat3D<0>` for the linear elastic case without history and `Umat3D<n>` for the case of history dimension n.

Note, assuming that the UMAT routine is available as Fortran code, the underlying C++ implementation of the `Umat3D` class follows the rules of mixed-language programming, namely: the call-by-address convention in Fortran, the row-major order of C++ arrays (contrary to Fortran's column-major order), and compiler-specific conventions such as appending underscores to Fortran subroutine names. For this paper, the `gfortran` and the `ifort` compilers are used. Care and attention are also required because of incompatible character pointers between Fortran and C++. The application of an external library of constitutive laws, as demonstrated by `labtools`, is showcased in Section 4.2. A similar methodology could be applied to incorporate laws implemented in alternative tools, such as USERMAT for the FE program Ansys.

4. Examples

The `fenics-constitutive` package continues to evolve by incorporating additional constitutive models. Currently, the available models include linear elasticity, J2 plasticity with nonlinear isotropic hardening and two standard linear solid models in both Maxwell representation and Kelvin–Voigt representation (e.g. [32]). To highlight the flexibility of the interface, we demonstrate its use for linear elasticity implemented in various programming languages, as well as for complex viscoplastic models available as UMATs.

4.1. Linear elasticity in Rust, C++, Python and UMAT

The arguably simplest material model – linear elasticity – is implemented using four different approaches as discussed in the previous sections. The material model mainly consists of one matrix–vector product:

$$\boldsymbol{\sigma}^{n+1} = \boldsymbol{\sigma}^{n} + \mathbb{C} \cdot\cdot \Delta\boldsymbol{\varepsilon}^{n}, \tag{19}$$

$$\frac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{\varepsilon}} = \mathbb{C}. \tag{20}$$

In the Rust, C++ and Python example (with `numpy`) the tangent $\mathbb{C}$ is stored as a class variable and can therefore be copied to the quadrature space without recalculating it in each evaluation of the constitutive model. For the UMAT, this is not possible since the UMAT only exists as a subroutine that cannot store any internal data and therefore the tangent is determined for each quadrature point, resulting in more computation time which is shown in Fig. 1(a). The Rust and C++ implementations show nearly identical timings. This is unsurprising since both use highly optimized linear algebra libraries and have the same basic structure with a for loop and one small matrix–vector product in each iteration. Additionally, both use the same compiler backend which can result in very similar machine code. The Python implementation does not implement a for-loop over the quadrature points directly, but works by directly multiplying the underlying storage vector of the strains (converted from the gradient of the displacement increment) with the tangent $\mathbb{C}$. This can result in higher memory consumption, but the `numpy` library is also highly optimized and therefore the runtime is the third-best.

The runtimes for the wrapped UMAT are generally the worst, which can be expected since the stresses and tangents need an extra conversion step between Mandel notation and Voigt notation, and the tangent is calculated for each quadrature point. However, for constitutive models that are more complex than a simple matrix–vector product, the expected overhead is negligible.

The overall runtime, including the solution of the linear system of equations, also shows that the linear elastic constitutive model, including the conversion between Voigt notation and Mandel notation for the UMAT, plays a minor part in the overall runtime since the timings for all implementations are similar.

4.2. External library of user-defined models labtools

`labtools` is a custom library of user-material subroutines (UMATs), a lightweight version of which is provided alongside the `fenics-constitutive` interface on Zenodo [29]. It can be found in the folder `/src/labtools`, with corresponding documentation available under `/src/labtools/doc`. Originally designed to enhance the implementation of crystal plasticity models, `labtools` facilitates UMAT maintenance and enables modular coding. It also provides tools for common tasks, such as reading temperature-dependent material parameters, explicit and implicit solvers for integrating ODEs and defining flow laws. Additionally, `labtools` includes a dedicated module for tensor operations. All these features simplify implementation tasks, make UMATs lightweight and readable, and make the library particularly useful for introducing custom material behaviors in Abaqus. `labtools` has been employed in various research studies [33–36].

Benchmark: extended Chaboche model. We consider the widely used model of Chaboche for polycrystals [37], which appropriately reproduces the rate-dependent mechanical response of the metallic components under high temperatures. The model employs evolution rules for the viscoplastic yielding, kinematic and isotropic hardening. The implementation in `labtools`, further referred to as `sdchaboX`, is extended for the second independent back stress variable in order to provide a more accurate description of hysteresis [38]

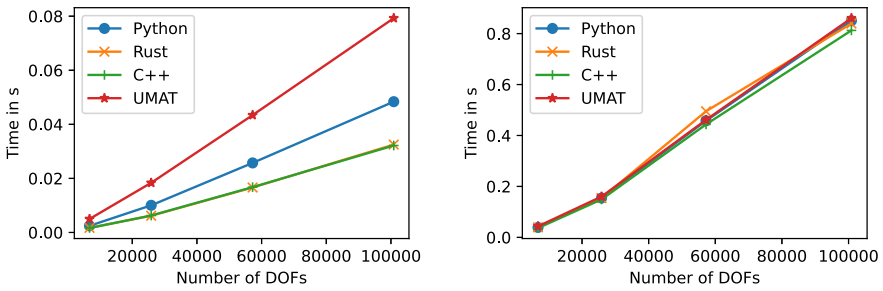
$$\chi = \sum \chi_i, \ i = 1, 2, \tag{21}$$

$$\dot{\chi}_i = \frac{2}{3} c\, \dot{\boldsymbol{\varepsilon}}^{vp} - d\, \chi_i\, \dot{p} - \left(\frac{J_2(\chi_i)}{M} \right)^{m} \frac{\chi_i}{J_2(\chi_i)}, \tag{22}$$

where χ is the total back stress and $\dot{p} = \sqrt{\frac{2}{3} \dot{\boldsymbol{\varepsilon}}^{vp} \cdot\cdot \dot{\boldsymbol{\varepsilon}}^{vp}}$ is the equivalent viscoplastic strain rate, c, d, m and M are model parameters. Each hardening evolves collinear with the viscoplastic strain rate $\dot{\boldsymbol{\varepsilon}}^{vp}$ and relieves due to the dynamic (proportional to $\dot{p}$) and static recoveries. The former is usually triggered by climb and cross-slip annihilation events whereas the latter acts on the larger time scale and governs, for instance, the steady-state creep.

The viscoplastic response of a notched plate was simulated under tensile loading along the Oy axis. A displacement of $10^{-3} L$ is applied to the top edge, $\partial\Omega_{\text{top}}$, of the specimen, as depicted in Fig. 2. Due to symmetry considerations, only one-fourth of the specimen is analyzed under plane strain conditions. Symmetric boundary conditions, $\bar{u}_y = 0$ and $\bar{u}_x = 0$, are enforced on the horizontal and vertical axes of symmetry. The FE model employs triangular elements with quadratic interpolation of the displacement field.

The mechanical problem's implementation into FEniCSx requires passing time increments to the class `IncrSmallStrainProblem` due to the rate-dependent nature of the applied constitutive law. Additionally, the update of history variables must occur whenever global structural equilibrium is reached. The provided code snippet in listing 4 addresses these requirements in a simplified manner.



(a) Constitutive model. (b) Total solve time.

Fig. 1. Timings for linear elasticity example. Mean of measurements over 1000 simulations for each number of DOFs.

```
# dynamic loading of the shared object into a module
module = importlib.util.module_from_spec(spec)

# create the constitutive law
law = module.Umat3D29(
  [],
  {
    "cmname":str(Path(__file__).parent / "SDCHABOX"),
    "libname":<Path to> / "labtools" / "lib" / "libumat.so"),
    "fEval":"kusdchabox_",
    "fParam":"param0_sdchabox_",
  }
  )

# assign the law to the set of cells 'body' and define BVP
laws = [(law, body)]
problem = IncrSmallStrainProblem(laws, u, [BCs], 2)

# define solver
solver = df.nls.petsc.NewtonSolver(MPI.COMM_WORLD, problem)

# create output
xdmf = df.io.XDMFFile(mesh.comm, "output.xdmf", "w")

while (AppliedLoading):
  # update BCs
  BCs.value = t * loadRate

  # update time increment in problem
  problem._del_t = t - problem._time

  # solve problem
  niter, converged = solver.solve(u)

  # update history variables
  problem.update()

  # update output
  xdmf.write_function(u, t)

  # update time
  t += dt
```

Listing 4: Principal scheme for implementation of a visco-plastic mechanical problem in FEniCSx.

First, the interface to UMAT, compiled to `umat.cpython-310-x86_64-linux-gnu.so`, see Section 3.3, is loaded as a module in line 2. The constitutive law that handles 29 history variables is then created and bound to `sdchaboX` model in the shared library `libumat.so`. The `kusdchabox_` subroutine provides the interface for the UMAT call, which reads the material parameters from `SDCHABOX.mat` file using the `param0_sdchabox_` routine. Then, the law is assigned to a set of cells (in this example, `body` includes all cells of the FE model), as shown in line 16, and passed to create the boundary value problem `IncrSmallStrainProblem`. The while-loop iterates over the time increments, updates the loading, calls the solver, updates all history variables of the constitutive law in all quadrature points of the structure (line 36), and writes to the output file. A convergence check and, if necessary, the implementation of adaptive time stepping is required when seeking global equilibrium in line 33. Since this is standard practice in incremental integrations and is beyond the scope of the benchmark, we omit it from the listing for clarity. The `solve(u)` method thus overwrites the displacement field $\boldsymbol{u}$. The instructions to run the script and the used values of the material parameters are described in the published paper code on Zenodo [29].

The distribution of the equivalent viscoplastic strain is demonstrated in Fig. 2. The reduced section results in a stress concentration which leads to enhanced yielding. Note, the visualization of the simulation results obtained by Abaqus and by FEniCSx is done by native viewers. That leads to slightly different colored maps, however, the results are equivalent within the computational tolerances. For instance, the reaction forces predicted by Abaqus and FEniCSx on $\partial\Omega_{\text{top}}$ are compared in Fig. 3(a). Similarly, the evolution of the equivalent viscoplastic strain in the highly-stressed point (its position is marked in Fig. 2) is the same for both simulations, see Fig. 3(b).

The constitutive equations are integrated in the Gauss points using the backward Euler discretization and the Newton–Raphson method with a line search procedure. The respective solver is implemented as a separate module in `labtools`. The computed stress tensor and consistent tangent stiffness are then transferred to the Newton–Raphson iterations within FEniCSx for finding the global equilibrium. The nearly quadratic convergence rate is observed in Fig. 3(c) for three selected time increments (the first increment with moderate yielding, the stabilized response at the last increment, and the transient response in the meantime). Note, Abaqus uses the largest residual force to control the accuracy of the solution, whereas FEniCSx operates with the Euclidean l^2-norm of the residuum.

Benchmark: polycrystalline aggregate. In the previous example, the constitutive model was assigned to all Gauss points of the FE structure. Here, we use `fenics-constitutive` to assign a specific constitutive law to a selected set of Gauss points. This flexibility is particularly advantageous for modeling multicomponent structures. In what follows, we leverage this capability to simulate the deformation of a polycrystalline aggregate. Although the single-crystal viscoplastic model is consistent across all grains, the grains differ in their crystallographic orientations. They influence the constitutive response and must

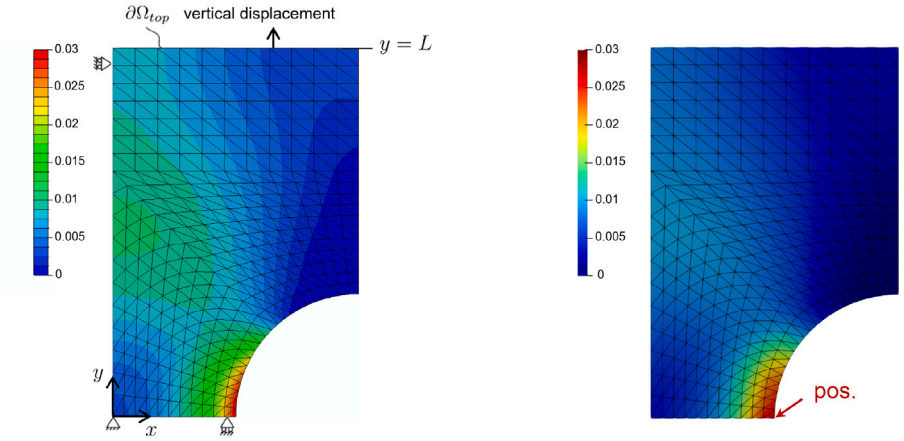


Fig. 2. Distribution of the viscoplastic equivalent strain over the notched sample: solutions obtained by Abaqus (left, visualized by Abaqus Viewer) and by FEniCSx (right, visualized by Paraview).

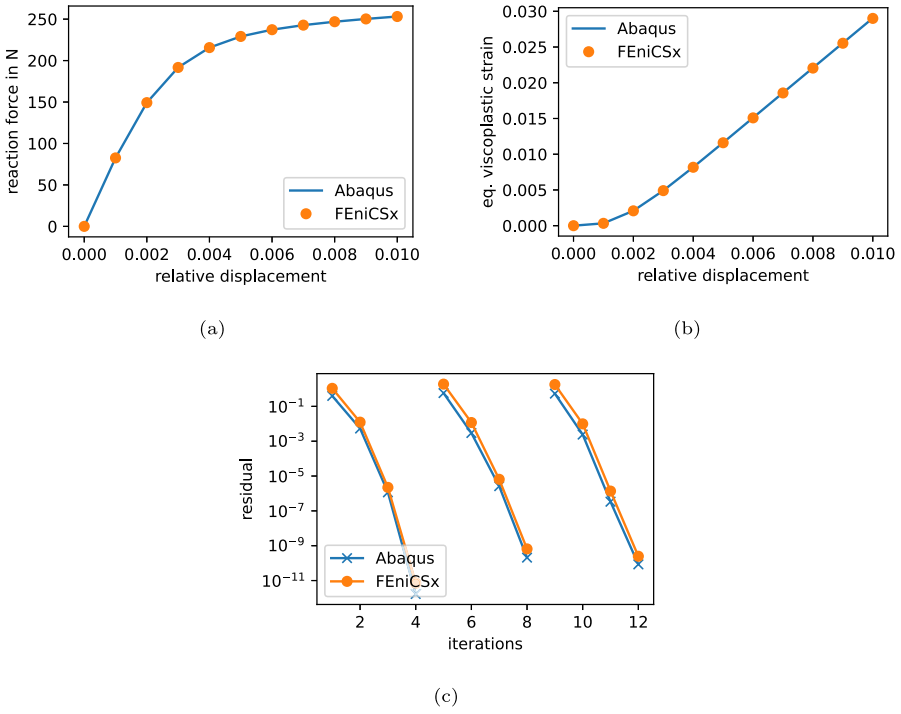


Fig. 3. (a) Reaction force, (b) the viscoplastic equivalent strain on the notch, and (c) the convergence analyses for the initial, an intermediate, and the last time increments.

be properly accounted for at each integration point. More precisely, we demonstrate how the constitutive model, including the specific crystallographic orientation, can be assigned to the cells (and thus to the Gauss points) associated with the respective grain.

An aggregate of 25 differently oriented grains, as shown in Fig. 4, undergoes creep deformation due to a load of 35 MPa applied to the top face in the Oz direction. The open-source software Neper [39] was used for tessellating and meshing the grains with linear tetrahedral elements. The FE model does not account for any deterioration or slip of the grain boundaries; therefore, the entire yielding is attributed to the viscoplastic flow of the grains. This is described by the crystallographic model proposed in [35], which simplifies the model of Meric et al. [40] for viscoplastic yielding in face-centered cubic single-crystals by discarding cubic slip, kinematic hardening, and interactions between slip systems. The micromechanical model based on this simplified constitutive relationship could reasonably simulate the creep of a copper-antimony alloy at 550 °C. For a detailed description of the model equations, which assume small strains, octahedral slip, Norton's flow law, and isotropic hardening, refer to the original paper [35]. Instructions for starting the simulations can be found in the published paper code on Zenodo [29].

First, the simulation using Abaqus was carried out with adaptive time stepping. The resulting time increments were then used for the simulation in FEniCSx to obtain comparable results. In the Abaqus model, the grains were defined by solid sections, which included respective element sets, a common constitutive model, and grain orientations. The groups of cells representing the grains can be imported into the FEniCSx model using the `meshio` tool [41], which identifies the mesh entities from the Abaqus input file. Care is needed because the cell numeration in FEniCSx differs from that in Abaqus; this can be found in the method `dolfinx.mesh.Mesh.topology.original_cell_index`. The constitutive laws with the proper crystallographic orientation are created by iterating over the cell groups. The series of laws and associated cell groups are passed to the mechanics problem, where

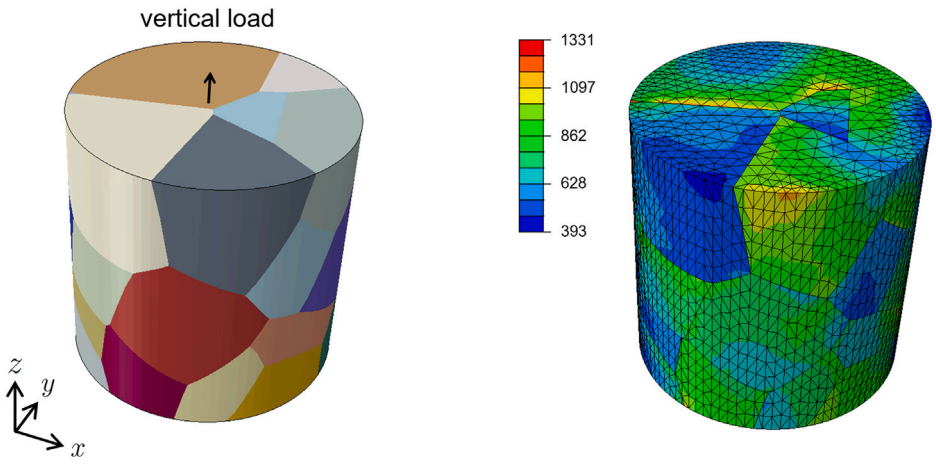
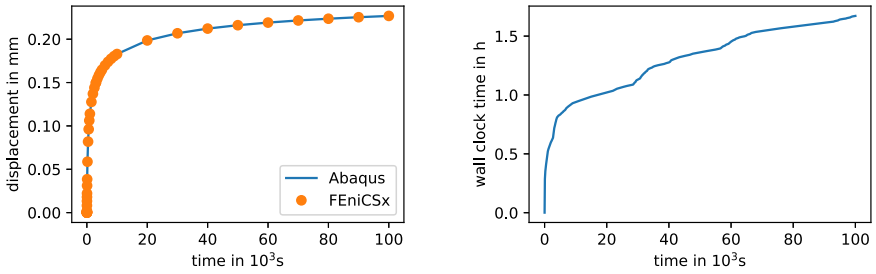


Fig. 4. The aggregate structure (colors mark the individual grains) and the distribution of the von Mises stress (on the right).



(a) (b)

Fig. 5. (a) Displacement on the top face of the aggregate subjected to creep loading and (b) computational efforts of the FE simulations by FEniCSx version 0.6.0.

the update of history variables occurs within the sub-meshes attributed to the cell groups. Note that `labtools` uses Euler angles to define the material coordinate system, a common practice in materials science. The benchmark script converts the orientations from the Abaqus format into Euler angles once the solid sections from the Abaqus input file are read. Since the constitutive response is rate-dependent, the implementation of the time integration follows the same scheme as in listing 4. Fig. 5(a) demonstrates that the simulation results match within the computational tolerances in the displacement on the top face, when computed by both Abaqus and FEniCSx.

The wall clock time of FEniCSx for the benchmark simulation of the polycrystalline aggregate is presented in Fig. 5(b). The `clang` compiler was used for C++, as set by default in the task file. An iterative Krylov solver was employed to solve the linear system of equations at each global Newton–Raphson iteration. The simulations were conducted using a single thread on a workstation equipped with an Intel Xeon Gold 6354 CPU operating at 3 GHz and 384 GB of RAM. Note that a direct one-to-one comparison with Abaqus performance is not reliable, as the exact solver settings, particularly the preconditioner setup, cannot be manually adjusted in Abaqus. However, the Abaqus simulations we conducted are stored in the repository and demonstrate a qualitatively similar wall clock time evolution to that of FEniCSx.

5. Conclusions

The open-source platform FEniCSx is widely used for solving partial differential equations, particularly in academic settings, due to its high level of abstraction and flexibility in handling the weak form of boundary value problems (BVPs). This paper presents an interface that unlocks FEniCSx's capabilities to structural mechanics, making the application and generic implementation of custom constitutive equations, including those with history variables, straightforward within the FEniCSx framework. The interface supports a multiple-language implementation strategy, provided that the code, which computes stresses and algorithmic material tangents, can be linked to Python. Users can directly benefit from our interface for implementing their constitutive laws in C++, Rust, and Python. The linear elasticity example demonstrates such implementations, showing nearly equivalent computational performance across the languages.

The underlying idea of how to integrate the sophisticated constitutive laws into FEniCSx originates from [14], where it was suggested to fill the vectors holding the values of the stress and stiffness tensors at each integration point when evaluating the weak form. More precisely, BVPs are implemented within the interface as a class derived from `dolfinx.fem.petsc.NonlinearProblem`. Each time the Newton solver method of the inherited class starts an iteration, the implemented `evaluate` method loops over the integration points and computes the history variables, stresses, and the algorithmic tangent matrix, or calls a respective external subroutine (like a user material subroutine (UMAT) from the commercial Abaqus software). After the solver's convergence, the FE coefficient vectors and the vector of history variables are updated in the associated quadrature spaces. Since custom constitutive laws can be assigned to the cells of specific sub-meshes, structural problems involving multi-material assemblies can be addressed, as demonstrated in the benchmark for a polycrystalline aggregate.

The binding of a pre-compiled library of UMATs is demonstrated as additional benchmarks. Since evaluating a UMAT requires conversion steps between Voigt notation and Mandel notation, the evaluation of the constitutive law is more time-consuming than with C++, Rust, or Python. However, the impact on the overall time for solving BVPs was insignificant. The results of FEniCSx simulations were successfully verified against the Abaqus program for nonlinear, rate-dependent constitutive laws with history variables.

Our contribution represents a step towards tool-independent implementation and manipulation of constitutive relationships, thereby advancing the development of standards for mechanical simulations. In the future, the interface has the potential to be extended for large deformation implementations and to address specific coupled problems as separate developments. As a further extension, we propose enhancing our interface with UFL external operators [20] and ensuring the constitutive model is derivable, enabling seamless integration into chain-rule derivations for derivative-based optimization and calibration (e.g., Hamiltonian Monte Carlo).

CRediT authorship contribution statement

Sjard Mathis Rosenbusch: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation. **Philipp Diercks:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation. **Vitaliy Kindrachuk:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation. **Jörg F. Unger:** Writing – review & editing.

Software availability

`fenics-constitutive` is available on Zenodo [13] and on GitHub.[5] The code used for the FEniCSx examples in this paper is published on Zenodo [29]. Only free open source software is needed to run the examples and the compute-environment is defined in a `conda-lock` file[6] to ensure reproducible results.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to check and improve the readability of the text. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Financial disclosure

None reported.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We want to thank Thomas Titscher who worked on a previous version of `fenics-constitutive` as well as on binding UMATs to legacy FEniCS. We want to thank our colleagues Bernard Fedelich and Cetin Haftaoglu, both from BAM, and Hans-Dieter Noack (retired, formerly with BAM) for the usage of a light version of `labtools`.

The authors would like to thank the Federal Government and the Heads of Government of the Länder, as well as the Joint Science Conference (GWK), for their funding and support within the framework of the NFDI4Ing consortium. Funded by the German Research Foundation (DFG) - project number 442146713.

The authors gratefully acknowledge the support of VDI Technologiezentrum GmbH and the Federal Ministry for Education and Research (BMBF) within the collaborative project "Lebenszyklus von Beton—Ontologieentwicklung für die Prozesskette der Betonherstellung."

Data availability

The data used to generate the graphs for the Abaqus examples is published on Zenodo [29].

[5] https://github.com/BAMresearch/fenics-constitutive

[6] https://github.com/conda/conda-lock

References

[1] Alnæs Martin, Blechta Jan, Hake Johan, Johansson August, Kehlet Benjamin, Logg Anders, et al. The FEniCS project version 1.5. Arch Numer Softw 2015;Vol 3. http://dx.doi.org/10.11588/ANS.2015.100.20553.

[2] Alnæs Martin S, Logg Anders, Ølgaard Kristian B, Rognes Marie E, Wells Garth N. Unified form language: A domain-specific language for weak formulations of partial differential equations. ACM Trans Math Software 2014;40(2):1–37. http://dx.doi.org/10.1145/2566630.

[3] Scroggs Matthew W, Baratta Igor A, Richardson Chris N, Wells Garth N. Basix: a runtime finite element basis evaluation library. J Open Source Softw 2022;7(73):3982. http://dx.doi.org/10.21105/joss.03982.

[4] Scroggs Matthew W, Dokken Jørgen S, Richardson Chris N, Wells Garth N. Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. ACM Trans Math Software 2022;48(2):1–23. http://dx.doi.org/10.1145/3524456.

[5] Baratta Igor A, Dean Joseph P, Dokken Jørgen S, Habera Michal, Hale Jack S, Richardson Chris N, et al. DOLFINx: The next generation fenics problem solving environment. Zenodo; 2023, http://dx.doi.org/10.5281/zenodo.10447666.

[6] Logg Anders, Wells Garth N. DOLFIN: Automated finite element computing. ACM Trans Math Software 2010;37(2):1–28. http://dx.doi.org/10.1145/1731022.1731030.

[7] Dhondt Guido, Wittig Klaus. CalculiX: A free software three-dimensional structural finite element program. 2024, URL https://calculix.de/.

[8] Africa Pasquale C, Arndt Daniel, Bangerth Wolfgang, Blais Bruno, Fehling Marc, Gassmöller Rene, et al. The deal.ii library, version 9.6. J Numer Math 2024;32(4):369–80. http://dx.doi.org/10.1515/jnma-2024-0137.

[9] Électricité de France (EDF). Code_aster – Finite element analysis for structural mechanics. 2001, URL https://www.code-aster.org/.

[10] Giudicelli Guillaume, Lindsay Alexander, Harbour Logan, Icenhour Casey, Li Mengnan, Hansel Joshua E, et al. 3.0 - MOOSE: Enabling massively parallel multiphysics simulations. SoftwareX 2024;26:101690. http://dx.doi.org/10.1016/j.softx.2024.101690, URL https://www.sciencedirect.com/science/article/pii/S235271102400061X.

[11] Malinen Mika, Råback Peter. Elmer finite element solver for multiphysics and multiscale problems. In: Kondov Ivan, Sutmann Godehart, editors. Multiscale modelling methods for applications in material science. Forschungszentrum Jülich; 2013, p. 101–13.

[12] Dassault Systèmes. SIMULIA user assistance: UMAT. 2023, URL https://help.3ds.com.

[13] Diercks Philipp, Robens-Radermacher Annika, Rosenbusch Sjard Mathis, Unger Jörg F, Saif-Ur-Rehman. fenics-constitutive. Zenodo; 2025, http://dx.doi.org/10.5281/zenodo.14882462.

[14] Bleyer Jeremy. Numerical tours of computational mechanics with FEniCS. Zenodo; 2018, http://dx.doi.org/10.5281/zenodo.1287832.

[15] Helfer Thomas, Michel Bruno, Proix Jean-Michel, Salvo Maxime, Sercombe Jérôme, Casella Michel. Introducing the open-source mfront code generator: Application to mechanical behaviours and material knowledge management within the PLEIADES fuel element modelling platform. Comput Math Appl 2015;70(5):994–1023. http://dx.doi.org/10.1016/j.camwa.2015.06.027.

[16] Bleyer Jeremy. dolfinx_materials: A Python package for advanced material modelling. Zenodo; 2024, http://dx.doi.org/10.5281/zenodo.13882183.

[17] Rosenbusch Sjard Mathis, Balzani Daniel, Unger Jörg F. Regularization of softening plasticity models for explicit dynamics using a gradient-enhanced modified Johnson–Holmquist model. Int J Impact Eng 2025;198:105209. http://dx.doi.org/10.1016/j.ijimpeng.2024.105209, URL https://www.sciencedirect.com/science/article/pii/S0734743X24003348.

[18] Latyshev Andrey, Bleyer Jérémy, Hale Jack, Maurini Corrado. A framework for expressing general constitutive models in fenicsx. In: CSMA 2024. February 2024.

[19] Latyshev Andrey, Hale Jack. dolfinx-external-operator: v.0.0.1. Zenodo; 2024, http://dx.doi.org/10.5281/zenodo.10907418.

[20] Latyshev Andrey, Bleyer Jérémy, Maurini Corrado, Hale Jack S. Expressing general constitutive models in FEniCSx using external operators and algorithmic automatic differentiation. 2024, working paper or preprint URL https://hal.science/hal-04735022.

[21] Bouziani Nacime, Ham David A. Escaping the abstraction: a foreign function interface for the unified form language [UFL]. 2021, CoRR abs/2111.00945 URL https://arxiv.org/abs/2111.00945.

[22] Lam Siu Kwan, Pitrou Antoine, Seibert Stanley. Numba: a LLVM-based Python JIT compiler. In: Proceedings of the second workshop on the LLVM compiler infrastructure in HPC. SC15, ACM; 2015-11, http://dx.doi.org/10.1145/2833157.2833162.

[23] Frostig Roy, Johnson Matthew, Leary Chris. Compiling machine learning programs via high-level tracing. In: SysML. 2018, URL https://mlsys.org/Conferences/doc/2018/146.pdf.

[24] Bertram Albrecht, Glüge Rainer. Solid mechanics: Theory, modeling, and problems. Springer International Publishing; 2015, http://dx.doi.org/10.1007/978-3-319-19566-7.

[25] Abali Bilen Emek. Computational reality. Springer Singapore; 2017, http://dx.doi.org/10.1007/978-981-10-2444-3,
[26] Hughes Thomas JR, Winget James. Finite rotation effects in numerical integration of rate constitutive equations arising in large-deformation analysis. Internat J Numer Methods Engrg 1980;15(12):1862–7. http://dx.doi.org/10.1002/nme.1620151210, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.1620151210 URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1620151210.
[27] Balay Satish, Abhyankar Shrirang, Adams Mark F, Benson Steven, Brown Jed, Brune Peter, et al. PETSc web page. 2024, URL https://petsc.org/.
[28] Mandel Jean. Generalisation de la theorie de plasticite de W. T. Koiter. Int J Solids Struct 1965;1(3):273–95. http://dx.doi.org/10.1016/0020-7683(65)90034-X, URL https://www.sciencedirect.com/science/article/pii/002076836590034X.
[29] Rosenbusch Sjard Mathis, Diercks Philipp, Kindrachuk Vitaliy, Unger Jörg. Integrating custom constitutive models into FEniCSx: A versatile approach and case studies. Zenodo; 2024, http://dx.doi.org/10.5281/zenodo.13980988.
[30] Jakob Wenzel. Nanobind: tiny and efficient C++/Python bindings. 2022, https://github.com/wjakob/nanobind.
[31] Guennebaud Gaël, Jacob Benoît, et al. Eigen v3. 2010, http://eigen.tuxfamily.org.
[32] Gross Dietmar, Hauger Werner, Wriggers Peter. Viskoelastizität und plastizität. In: Technische Mechanik 4: Hydromechanik, Elemente der Höheren Mechanik, Numerische Methoden. Berlin, Heidelberg: Springer Berlin Heidelberg; 2023, p. 325–406. http://dx.doi.org/10.1007/978-3-662-66524-4_6.
[33] Gruetzner Stefan, Fedelich Bernard, Rehmer Birgit, Buchholz Bjoern. Material modelling and lifetime prediction of Ni-base gas turbine blades under TMF conditions. In: 11th international fatigue congress. Advanced materials research, vol. 891, Trans Tech Publications Ltd; 2014, p. 1277–82. http://dx.doi.org/10.4028/www.scientific.net/AMR.891-892.1277.
[34] Kindrachuk Vitaliy M, Galanov Boris A. An efficient approach for numerical treatment of some inequalities in solid mechanics on examples of Kuhn–Tucker and Signorini–Fichera conditions. J Mech Phys Solids 2014;63:432–50. http://dx.doi.org/10.1016/j.jmps.2013.08.008.
[35] Vöse Markus, Otto Frederik, Fedelich Bernard, Eggeler Gunther. Micromechanical investigations and modelling of a Copper–Antimony-Alloy under creep conditions. Mech Mater 2014;69(1):41–62. http://dx.doi.org/10.1016/j.mechmat.2013.09.013.
[36] Gesell Stephan, Ganesh Rahul, Kuna Meinhard, Fedelich Bernard, Kiefer Bjoern. Numerical calculation of ΔCTOD to simulate fatigue crack growth under large scale viscoplastic deformations. Eng Fract Mech 2023;281:109064. http://dx.doi.org/10.1016/j.engfracmech.2023.109064, URL https://www.sciencedirect.com/science/article/pii/S001379442300022X.
[37] Chaboche Jean-Louis. A review of some plasticity and viscoplasticity constitutive theories. Int J Plast 2008;24(10):1642–93. http://dx.doi.org/10.1016/j.ijplas.2008.03.009, Special Issue in Honor of Jean-Louis Chaboche.
[38] Sommitsch Christof, Sievert Rainer, Wlanis Thomas, Günther Burkhard, Wieser Volker. Modelling of creep-fatigue in containers during aluminium and copper extrusion. Comput Mater Sci 2007;39(1):55–64. http://dx.doi.org/10.1016/j.commatsci.2006.03.024, Proceedings of the 15th International Workshop on Computational Mechanics of Materials.
[39] Quey Romain, Dawson Paul R, Barbe Fabrice. Large-scale 3D random polycrystals for the finite element method: Generation, meshing and remeshing. Comput Methods Appl Mech Engrg 2011;200(17):1729–45. http://dx.doi.org/10.1016/j.cma.2011.01.002.
[40] Méric Laurent, Poubanne Philippe, Cailletaud Georges. Single crystal modeling for structural calculations: Part 1—Model presentation. J Eng Mater Technology-Trans the Asme 1991;113:162–70.
[41] Schlömer Nico. meshio: Tools for mesh files. Zenodo; 2024, http://dx.doi.org/10.5281/zenodo.1288334.