Advanced material modeling in `FEniCSx`

Jérémy Bleyer
coll.: Andrey Latyshev, Corrado Maurini, Jack S. Hale

Laboratoire Navier, ENPC, Univ Gustave Eiffel, CNRS

MFront User Days
November 19th 2024

`http://fenicsproject.org/`

collection of free, open source, software components for **automated solution** of differential equations

Features:

- automated solution of variational formulation (same spirit as FreeFem++, deal.ii, etc.)
- extensive library of finite elements
- designed for parallel computation (high-performance linear algebra through PETSc backends)
- simple Python interface and concise high-level language, efficient C code generation

`http://fenicsproject.org/`

collection of free, open source, software components for **automated solution** of differential equations

Features:

- automated solution of variational formulation (same spirit as FreeFem++, deal.ii, etc.)
- extensive library of finite elements
- designed for parallel computation (high-performance linear algebra through PETSc backends)
- simple Python interface and concise high-level language, efficient C code generation

Applications:

- applied mathematics, fluid mechanics
- **solid mechanics, multiphysics** (heat transfer, transport, chemical reactions)
- electromagnetism, general relativity, ...

Non-linear problems

Finite-strain: Total Lagrangian formulation $\boldsymbol{P}$: 1st Piola-Kirchhoff stress

$$\int_\Omega \boldsymbol{P}(\boldsymbol{u}) : \nabla \boldsymbol{v} \,\mathrm{d}\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \,\mathrm{d}\Omega + \int_{\partial\Omega_\mathbf{N}} \boldsymbol{T} \cdot \boldsymbol{v} \,\mathrm{dS} \quad \forall \boldsymbol{v} \in V_0$$

Hyperelasticity: behavior derives from an elastic free energy $\psi(\boldsymbol{F})$ depending on the deformation gradient $\boldsymbol{F}(\boldsymbol{X}) = \boldsymbol{I} + \nabla_{\boldsymbol{X}} \boldsymbol{u}(\boldsymbol{X})$

Non-linear problems

Finite-strain: Total Lagrangian formulation $\boldsymbol{P}$: 1st Piola-Kirchhoff stress

$$\int_\Omega \boldsymbol{P}(\boldsymbol{u}) : \nabla \boldsymbol{v} \, \mathrm{d}\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \, \mathrm{d}\Omega + \int_{\partial\Omega_N} \boldsymbol{T} \cdot \boldsymbol{v} \, \mathrm{dS} \quad \forall \boldsymbol{v} \in V_0$$

Hyperelasticity: behavior derives from an elastic free energy $\psi(\boldsymbol{F})$ depending on the deformation gradient $\boldsymbol{F}(\boldsymbol{X}) = \boldsymbol{I} + \nabla_{\boldsymbol{X}} \boldsymbol{u}(\boldsymbol{X})$
Optimality conditions of the minimization problem:

$$\min_{\boldsymbol{u}\in V} \int_\Omega \psi(\boldsymbol{F}) \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} \, \mathrm{d}\Omega - \int_{\partial\Omega_N} \boldsymbol{T} \cdot \boldsymbol{u} \, \mathrm{dS}$$

$$\text{residual} \quad R(\boldsymbol{u}) = \int_\Omega \frac{\partial \psi}{\partial \boldsymbol{F}} : \nabla \boldsymbol{v} \, \mathrm{d}\Omega - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} \, \mathrm{d}\Omega - \int_{\partial\Omega_N} \boldsymbol{T} \cdot \boldsymbol{u} \, \mathrm{dS} = 0$$

$$\text{tangent operator} \quad K_{\text{tang}}(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \nabla \boldsymbol{u} : \frac{\partial^2 \psi}{\partial \boldsymbol{F} \partial \boldsymbol{F}} : \nabla \boldsymbol{v} \, \mathrm{d}\Omega$$

solvers: built-in Newton or PETSc SNES

dolfinx_materials: Python package for material behaviors

Objective: provide simple way of defining and handling complex material constitutive behaviors within `dolfinx`

dolfinx_materials: Python package for material behaviors

Objective: provide simple way of defining and handling complex material constitutive behaviors within `dolfinx`

Concept: see the constitutive relation as a *black-box function* mapping **gradients** (e.g. strain $\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u}$) to **fluxes** (e.g. stresses $\boldsymbol{\sigma}$) at the level of **quadrature points**

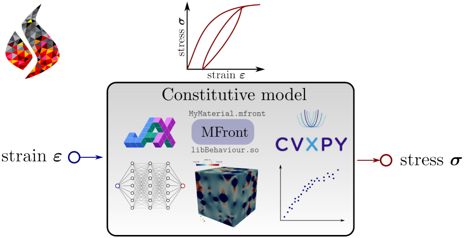
dolfinx_materials: Python package for material behaviors

Objective: provide simple way of defining and handling complex material constitutive behaviors within `dolfinx`

Concept: see the constitutive relation as a *black-box function* mapping **gradients** (e.g. strain $\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u}$) to **fluxes** (e.g. stresses $\boldsymbol{\sigma}$) at the level of **quadrature points**

Concrete implementation of the constitutive relation

- a user-defined Python function
- provided by an external library (e.g. behaviors compiled with MFront)
- a neural network inference
- solution to a FE computation on a RVE, etc.



A Python elasto-plastic behaviour

`Material`: provides info at the quadrature point level e.g. dimension of gradient inputs/stress outputs, stored internal state variables, required external state variables

```python
class ElastoPlasticIsotropicHardening(Material):
    @property
    def internal_state_variables(self):
        return {"p": 1} # cumulated plastic strain

    def constitutive_update(self, eps, state):
        eps_old = state["Strain"]
        deps = eps - eps_old
        p_old = state["p"]

        C = self.elastic_model.compute_C()
        sig_el = state["Stress"] + C @ deps    # elastic predictor
        s_el = K() @ sig_el
        sig_Y_old = self.yield_stress(state["p"])
        sig_eq_el = np.sqrt(3 / 2.0) * np.linalg.norm(s_el)
        if sig_eq_el - sig_Y_old >= 0:
            dp = fsolve(lambda dp: sig_eq_el - 3*mu*dp - self.yield_stress(p_old + dp), 0.0)
        else:
            dp = 0
        state["Strain"] = eps_old + deps
        state["p"] += dp
        return sig_el - 3 * mu * s_el / sig_eq_el * dp
```

Pseudo-code on the `dolfinx` side

`QuadratureMap`: storage of different quantities as `Quadrature` functions, evaluates UFL expression at quadrature points and material behavior for a set of cells

```
u = fem.Function(V)
qmap = QuadratureMap(u, deg_quad, material) # material = ["Strain"] --> ["Stress"]
qmap.register_gradient("Strain", eps(u))

sig = qmap.fluxes["Stress"]   # a function defined on "Quadrature" space

Res = ufl.inner(sig, eps(v)) * qmap.dx - ufl.inner(f, u) * dx
Jac = ...

for i in Newton_loop:       # custom Newton solver
    qmap.update()           # update current stress estimate
    b = assemble_vector(Res)
    A = assemble_matrix(Jac)
    solve(A, b, du.vector)  # compute displacement correction
    u.vector[:] += du.vector[:]

qmap.advance()              # updates previous state with current one for next time step
```

Above code **independent from** the `material`, provided that `gradients = ["Strain"]` and `fluxes = ["Stress"]`

About the Jacobian and non-linear solvers

`Material` should provide a "tangent" operator

```python
def constitutive_update(self, eps, state):
    [...]
    return sig, Ct
```

can be the algorithmic consistent operator, the secant, the elastic operator, etc...

```python
Res = ufl.inner(sig, eps(v)) * qmap.dx - ufl.inner(f, u) * dx
Jac = qmap.derivative(Res, u, du)
```

About the Jacobian and non-linear solvers

`Material` should provide a "tangent" operator

```
def constitutive_update(self, eps, state):
    [...]
    return sig, Ct
```

can be the algorithmic consistent operator, the secant, the elastic operator, etc...

```
Res = ufl.inner(sig, eps(v)) * qmap.dx - ufl.inner(f, u) * dx
Jac = qmap.derivative(Res, u, du)
```

Here: `qmap.derivative(Res, u, du) = ufl.derivative(Res, u, du) + ufl.inner(Ct * eps(du), eps(v)) * qmap.dx + ...` where `Ct` is a Quadrature function storing the values of $\dfrac{\mathtt{d"Stress"}}{\mathtt{d"Strain"}}$.

Available solvers: `NewtonSolver`, `PETSc.SNES`

`FEniCSx`/`MFront` integration

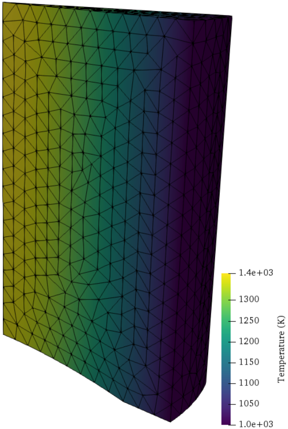
`MFrontMaterial` class for loading a MFront library, calling the behaviour integration and giving access to fluxes, state variables and tangent operators

The **only** metadata not provided by **MGIS** is how the gradients (e.g. strain) are expressed as functions of the unknown fields $\boldsymbol{u}$ (e.g. displacement)
The user is required to provide this link with UFL expressions (**registration**):

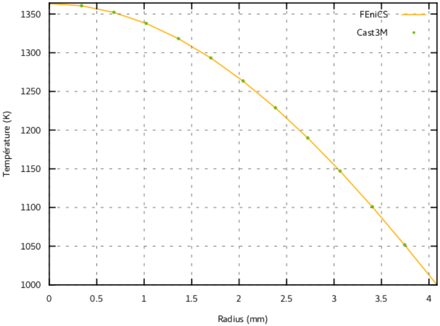
```
mat_prop = {"YoungModulus": E, "PoissonRatio": nu,
            "HardeningSlope": H, "YieldStrength": sig0}
material = MFrontMaterial("src/libBehaviour.so",
               "IsotropicLinearHardeningPlasticity",
               hypothesis="plane_strain",
               material_properties=mat_prop)

qmap = QuadratureMap(domain, deg_quad, material)
qmap.register_gradient("Strain", strain(u))
sig = qmap.fluxes["Stress"]

Res = ufl.dot(sig, strain(v)) * qmap.dx
Jac = qmap.derivative(Res, u, du)
```

DEMO

Examples - Stationary non-linear heat transfer



`quad_deg`	`dolfinx/MFront`	`dolfinx`
2	15.76 s	15.22 s
5	16.53 s	15.56 s



Multiphysics model for 3D concrete printing

$$\mathrm{d}\boldsymbol{\sigma} = \mathbb{C} : \mathrm{d}\boldsymbol{\varepsilon} - bS_\ell \mathrm{d}p\boldsymbol{I} - 3\alpha K \mathrm{d}T\boldsymbol{I}$$

$$\mathrm{d}\phi = b\,\mathrm{tr}(\mathrm{d}\boldsymbol{\varepsilon}) + \frac{b-\phi_0}{K_s}\mathrm{d}p - 3\alpha(b-\phi_0)\mathrm{dT}$$

$$\mathrm{d}S_s = 3\alpha K\,\mathrm{tr}(\boldsymbol{\varepsilon}) - 3\alpha(b-\phi_0)\mathrm{d}p + C\frac{1-\phi_0}{T_0}\mathrm{d}T$$

[Image: XtreeE]

Multiphysics model for 3D concrete printing

$$
\begin{aligned}
\mathrm{d}\boldsymbol{\sigma} &= \mathbb{C}(\xi) : \mathrm{d}\boldsymbol{\varepsilon} - b(\xi) S_\ell \mathrm{d}p \boldsymbol{I} - 3\alpha K(\xi) \mathrm{d}T \boldsymbol{I} \\
\mathrm{d}\phi &= b(\xi)\,\mathrm{tr}(\mathrm{d}\boldsymbol{\varepsilon}) + \frac{b(\xi) - \phi_0(\xi)}{K_s} \mathrm{d}p - 3\alpha(b(\xi) - \phi_0(\xi))\mathrm{dT} - \sum_{i=1}^{2} \Delta V_{s,i} \mathrm{d}\xi_i \\
\mathrm{d}S_s &= 3\alpha K(\xi)\,\mathrm{tr}(\boldsymbol{\varepsilon}) - 3\alpha(b(\xi) - \phi_0(\xi))\mathrm{d}p + C\frac{1 - \phi_0(\xi)}{T_0} \mathrm{d}T + \sum_{i=1}^{2} \frac{\mathcal{L}_i}{T_0} \mathrm{d}\xi_i
\end{aligned}
$$

Evolution of material properties with hydration + **change of solid volume** due to chemical reaction(s) + **heat** induced by reaction(s)

Multiphysics model for 3D concrete printing

$$\mathrm{d}\boldsymbol{\sigma} = \mathbb{C}(\xi) : \mathrm{d}\boldsymbol{\varepsilon} - b(\xi) S_\ell \mathrm{d}p \boldsymbol{I} - 3\alpha K(\xi) \mathrm{d}T \boldsymbol{I}$$

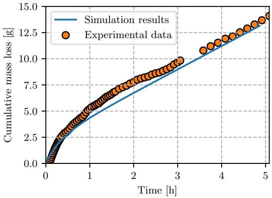
$$\mathrm{d}\phi = b(\xi)\,\mathrm{tr}(\mathrm{d}\boldsymbol{\varepsilon}) + \frac{b(\xi) - \phi_0(\xi)}{K_s}\mathrm{d}p - 3\alpha(b(\xi) - \phi_0(\xi))\mathrm{d}T - \sum_{i=1}^{2} \Delta V_{s,i} \mathrm{d}\xi_i$$

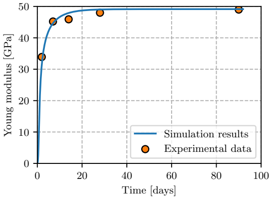
$$\mathrm{d}S_s = 3\alpha K(\xi)\,\mathrm{tr}(\boldsymbol{\varepsilon}) - 3\alpha(b(\xi) - \phi_0(\xi))\mathrm{d}p + C\frac{1 - \phi_0(\xi)}{T_0}\mathrm{d}T + \sum_{i=1}^{2} \frac{\mathcal{L}_i}{T_0}\mathrm{d}\xi_i$$

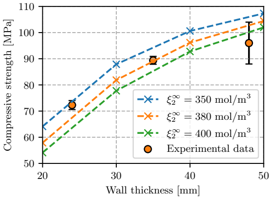
Evolution of material properties with hydration + **change of solid volume** due to chemical reaction(s) + **heat** induced by reaction(s)

[Maxime Pierre, Navier]: **Cam-Clay poroplasticity** from fresh to hardened state

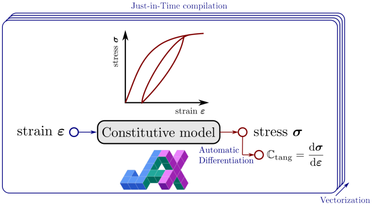
[Alice Gribonval, Navier]: influence of **environmental conditions** on compressive strength







JAX for constitutive modeling



JAX = accelerated (GPU) array computation and program transformation, designed for HPC and large-scale **machine learning**

```
def constitutive_update(eps, state, dt):
    [...]
```

- **JIT** and **automatic vectorization**

```
batch_constitutive_update = jax.jit(jax.vmap(constitutive_update, in_axes=(0, 0, None))
```

- **Automatic Differentiation**

```
constitutive_update_tangent = jax.jacfwd(constitutive_update, argnums=0, has_aux=True)
```

Mohr-Coulomb plasticity with apex smoothing using JAX [Latyshev et al., 2024]

Conclusions

Project available at:

`https://github.com/bleyerj/dolfinx_materials`

Library **currently supports**:

- `MFront` behaviors
- native `Python` behaviors (**slow**)
- `JAX` Python-like behaviors with **Automatic Differentiation**, see other demos
- **convex-optimization** based formulation using `cvxpy`

Upcoming features:

- **neural networks** demos
- **more extensive JAX behaviors**
- merge with `ExternalOperator` developments in `UFL` and `dolfinx` [Latyshev]
- model-free data-driven behaviors ?

Conclusions

Project available at:

`https://github.com/bleyerj/dolfinx_materials`

Library **currently supports**:

- `MFront` behaviors
- native `Python` behaviors (**slow**)
- `JAX` Python-like behaviors with **Automatic Differentiation**, see other demos
- **convex-optimization** based formulation using `cvxpy`

Upcoming features:

- **neural networks** demos
- **more extensive JAX behaviors**
- merge with `ExternalOperator` developments in `UFL` and `dolfinx` [Latyshev]
- model-free data-driven behaviors ?

Thank you for your attention !