Advanced material modeling in FEniCSx

Jérémy Bleyer
coll.: Andrey Latyshev, Corrado Maurini, Filippo Masi

Laboratoire Navier, ENPC, Univ Gustave Eiffel, CNRS

FEniCS 2023
June 14th-16th 2023

Nonlinear solid mechanics and nonlinear constitutive behavior

FEniCS archetypal example: **hyperelasticity**, e.g. compressible neo-Hookean

$$\psi(\boldsymbol{F}) = \frac{\lambda}{2}(J-1)^2 + \frac{\mu}{2}\left(I_1 - 3 - 2\ln J\right)$$

where $I_1 = \text{tr}(\boldsymbol{C}) = \text{tr}(\boldsymbol{F}^{\mathsf{T}}\boldsymbol{F})$ and $J = \det \boldsymbol{F}$.
Easy definition and derivation with **UFL operators**

Nonlinear solid mechanics and nonlinear constitutive behavior

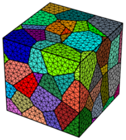
FEniCS archetypal example: **hyperelasticity**, e.g. compressible neo-Hookean

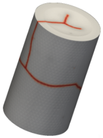
$$\psi(\boldsymbol{F}) = \frac{\lambda}{2}(J-1)^2 + \frac{\mu}{2}(I_1 - 3 - 2\ln J)$$

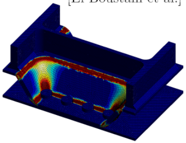
where $I_1 = \operatorname{tr}(\boldsymbol{C}) = \operatorname{tr}(\boldsymbol{F}^\mathsf{T}\boldsymbol{F})$ and $J = \det \boldsymbol{F}$.
Easy definition and derivation with **UFL operators**

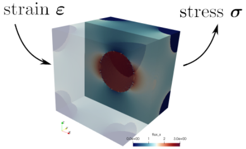
What about ?

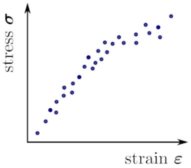
- **damage**, **viscoelasticity**, **(visco)plasticity**, etc.
- **multiscale** models
- **data-driven** models


[Neper/FEPX]

[Helfer et al.]


[El Boustani et al.]






Non-linear mechanics setting

Generic (small strain) setting: Find $\boldsymbol{u} \in V$ such that:

$$\int_\Omega \boldsymbol{\sigma}(\nabla^s \boldsymbol{u}) : \nabla^s \boldsymbol{v} \, d\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \, d\Omega + \int_{\partial\Omega_\mathbf{N}} \boldsymbol{T} \cdot \boldsymbol{v} \, dS \quad \forall \boldsymbol{v} \in V \tag{1}$$

Local non-linear mapping, **not expressible** using UFL

$$\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u} \longrightarrow \boxed{\text{CONSTITUTIVE RELATION}} \longrightarrow \boldsymbol{\sigma}$$

Non-linear mechanics setting

Generic (small strain) setting: Find $\boldsymbol{u} \in V$ such that:

$$\int_\Omega \boldsymbol{\sigma}(\nabla^s \boldsymbol{u}) : \nabla^s \boldsymbol{v} \, d\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \, d\Omega + \int_{\partial\Omega_\mathbf{N}} \boldsymbol{T} \cdot \boldsymbol{v} \, dS \quad \forall \boldsymbol{v} \in V \tag{1}$$

Local non-linear mapping, **not expressible** using UFL

$$\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u}, \mathcal{S}_n \longrightarrow \boxed{\text{CONSTITUTIVE RELATION}} \longrightarrow \boldsymbol{\sigma}, \mathcal{S}_{n+1}$$

- implicit non-linear equation
- implicit non-linear equations with state variables $\mathcal{S}_n$
- non-linear FE computation on a RVE
- Neural-Network inference
- closest-point projection onto a data manifold

Non-linear mechanics setting

Generic (small strain) setting: Find $\boldsymbol{u} \in V$ such that:

$$\int_\Omega \boldsymbol{\sigma}(\nabla^s \boldsymbol{u}) : \nabla^s \boldsymbol{v} \, \mathrm{d}\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \, \mathrm{d}\Omega + \int_{\partial\Omega_\mathbf{N}} \boldsymbol{T} \cdot \boldsymbol{v} \, \mathrm{d}S \quad \forall \boldsymbol{v} \in V \tag{1}$$

Local non-linear mapping, **not expressible** using UFL

$$\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u}, \mathcal{S}_n \longrightarrow \boxed{\text{CONSTITUTIVE RELATION}} \longrightarrow \boldsymbol{\sigma}, \mathcal{S}_{n+1}$$

- implicit non-linear equation
- implicit non-linear equations with state variables $\mathcal{S}_n$
- non-linear FE computation on a RVE
- Neural-Network inference
- closest-point projection onto a data manifold

Material modeling

a science in its own right, sometimes with dedicated tools

Additional challenges

- **Derivatives**: Newton method for solving (1) requires the **Jacobian**, at least:

$$\delta\boldsymbol{\sigma}(\nabla^s\boldsymbol{u}) = \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u}$$

Additional challenges

- **Derivatives**: Newton method for solving (1) requires the **Jacobian**, at least:

$$\delta\boldsymbol{\sigma}(\nabla^s \boldsymbol{u}) = \frac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{\varepsilon}} : \nabla^s \delta \boldsymbol{u}$$

  sometimes:

$$\begin{aligned}\delta\boldsymbol{\sigma}(\nabla^s \boldsymbol{u}, T) &= \frac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{\varepsilon}} : \nabla^s \delta \boldsymbol{u} + \frac{\partial \boldsymbol{\sigma}}{\partial T}\delta T \\ \delta\mathcal{S}_{n+1}(\nabla^s \boldsymbol{u}, T) &= \frac{\partial \mathcal{S}_{n+1}}{\partial \boldsymbol{\varepsilon}} : \nabla^s \delta \boldsymbol{u} + \frac{\partial \mathcal{S}_{n+1}}{\partial T}\delta T\end{aligned}$$

Additional challenges

- **Derivatives**: Newton method for solving (1) requires the **Jacobian**, at least:

$$\delta\boldsymbol{\sigma}(\nabla^s\boldsymbol{u}) = \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u}$$

  sometimes:

$$\begin{aligned}\delta\boldsymbol{\sigma}(\nabla^s\boldsymbol{u}, T) &= \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u} + \frac{\partial\boldsymbol{\sigma}}{\partial T}\delta T \\ \delta\mathcal{S}_{n+1}(\nabla^s\boldsymbol{u}, T) &= \frac{\partial\mathcal{S}_{n+1}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u} + \frac{\partial\mathcal{S}_{n+1}}{\partial T}\delta T\end{aligned}$$

- **Generalized stresses**: $\boldsymbol{\sigma}$ (solids), (N, M, Q) (beams), $(\boldsymbol{N}, \boldsymbol{M}, \boldsymbol{Q})$ (plates/shells), Cosserat, strain gradient, etc.

Additional challenges

- **Derivatives**: Newton method for solving (1) requires the **Jacobian**, at least:

$$\delta\boldsymbol{\sigma}(\nabla^s\boldsymbol{u}) = \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u}$$

  sometimes:

$$\begin{aligned}\delta\boldsymbol{\sigma}(\nabla^s\boldsymbol{u}, T) &= \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u} + \frac{\partial\boldsymbol{\sigma}}{\partial T}\delta T \\ \delta\mathcal{S}_{n+1}(\nabla^s\boldsymbol{u}, T) &= \frac{\partial\mathcal{S}_{n+1}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u} + \frac{\partial\mathcal{S}_{n+1}}{\partial T}\delta T\end{aligned}$$

- **Generalized stresses**: $\boldsymbol{\sigma}$ (solids), (N, M, Q) (beams), $(\boldsymbol{N}, \boldsymbol{M}, \boldsymbol{Q})$ (plates/shells), Cosserat, strain gradient, etc.
- **Coupled physics**: THM

Additional challenges

- **Derivatives**: Newton method for solving (1) requires the **Jacobian**, at least:

$$\delta\boldsymbol{\sigma}(\nabla^s\boldsymbol{u}) = \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u}$$

  sometimes:

$$\begin{aligned}\delta\boldsymbol{\sigma}(\nabla^s\boldsymbol{u}, T) &= \frac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u} + \frac{\partial\boldsymbol{\sigma}}{\partial T}\delta T \\ \delta\mathcal{S}_{n+1}(\nabla^s\boldsymbol{u}, T) &= \frac{\partial\mathcal{S}_{n+1}}{\partial\boldsymbol{\varepsilon}} : \nabla^s\delta\boldsymbol{u} + \frac{\partial\mathcal{S}_{n+1}}{\partial T}\delta T\end{aligned}$$

- **Generalized stresses**: $\boldsymbol{\sigma}$ (solids), (N, M, Q) (beams), $(\boldsymbol{N}, \boldsymbol{M}, \boldsymbol{Q})$ (plates/shells), Cosserat, strain gradient, etc.
- **Coupled physics**: THM
- **Finite strains**

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**

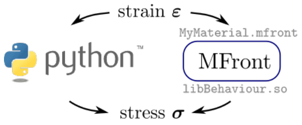
`dolfinx_materials`: Python package for material behaviors

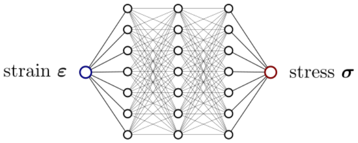
Objective: provide simple way of defining and handling complex material constitutive behaviors within `dolfinx`

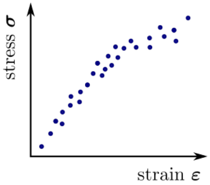
Concept: see the constitutive relation as a *black-box function* mapping **gradients** (e.g. strain $\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u}$) to **fluxes** (e.g. stresses $\boldsymbol{\sigma}$) at the level of **quadrature points**

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.


strain $\boldsymbol{\varepsilon}$
MyMaterial.mfront
python
MFront
libBehaviour.so
stress $\boldsymbol{\sigma}$


strain $\boldsymbol{\varepsilon}$
stress $\boldsymbol{\sigma}$



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

```
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)
```

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...

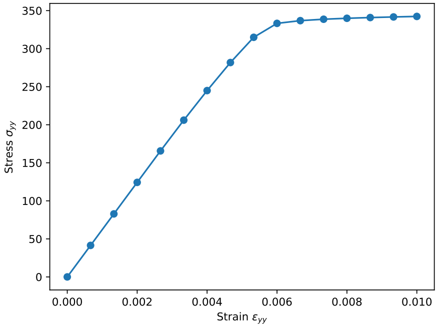
```python
Res = ufl.inner(sig, eps(v)) * qmap.dx - ufl.inner(f, u) * dx
Jac = qmap.derivative(Res, u, du)
```

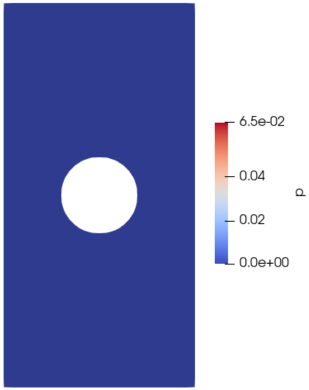
Here: `qmap.derivative(Res, u, du) = ufl.derivative(Res, u, du) + ufl.inner(Ct * eps(du), eps(v)) * qmap.dx + ...` where `Ct` is a Quadrature function storing the values of $\dfrac{\texttt{d"Stress"}}{\texttt{d"Strain"}}$.

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

Example

2D **plane strain** perforated plate in tension





Plane stress version

3D constitutive behaviour, 2D displacement $\boldsymbol{u}$:

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \nabla^s \boldsymbol{u} & 0 \\ 0 & \epsilon_{zz} \end{bmatrix}$$

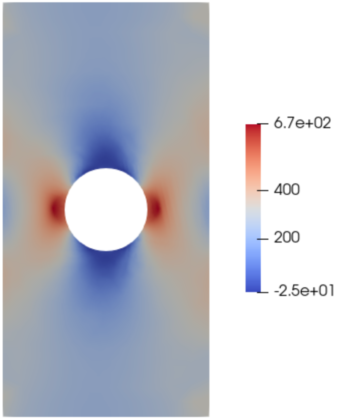
$$\boldsymbol{\sigma} = \begin{bmatrix} \boldsymbol{\sigma}_{2D} & 0 \\ 0 & 0 \end{bmatrix}$$

Mixed approach : Find $(\boldsymbol{u}, \epsilon_{zz}) \in V$ s.t.:

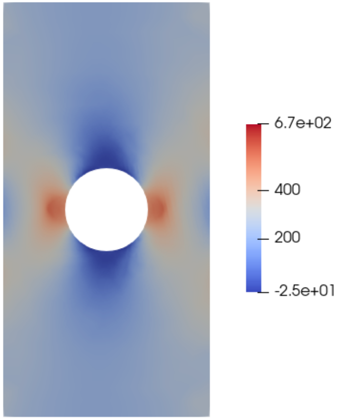
$$\int_\Omega \left(\boldsymbol{\sigma}_{2D}(\nabla^s \boldsymbol{u}, \epsilon_{zz}) : \nabla^s \boldsymbol{v} + \sigma_{zz}(\nabla^s \boldsymbol{u}, \epsilon_{zz})\widehat{\epsilon}_{zz}\right) \,\mathrm{d}\Omega = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{v}\,\mathrm{d}\Omega + \int_{\partial\Omega_{\mathbf{N}}} \boldsymbol{T}\cdot\boldsymbol{v}\,\mathrm{d}S \quad \forall(\boldsymbol{v}, \widehat{\epsilon}_{zz}) \in V$$

here $V =$`("CG", 2) x ("DG", 1)`

Plane stress version



(a) plane strain

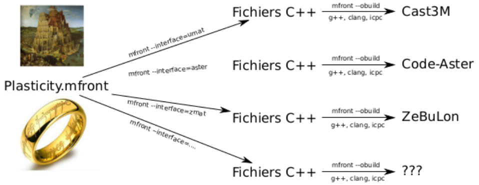


(b) plane stress

Figure: Vertical stress σ_{yy}

MFront: a code-generator tool for complex constitutive laws

developed at CEA, mainly by Thomas Helfer



Philosophy

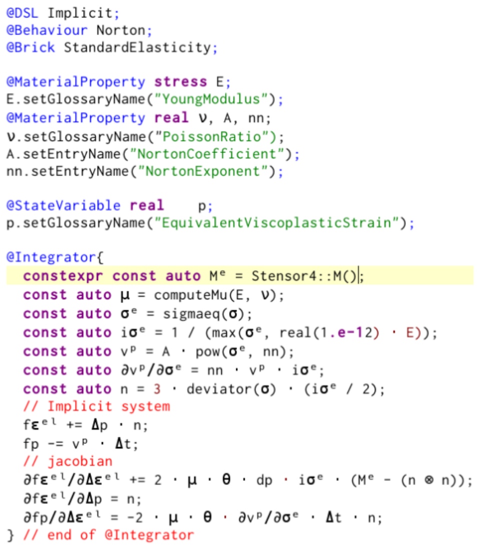
efficient, reliable, sustainable material library
independent of any FE solver:

- dedicated optimized interfaces (Abaqus, Castem, Code Aster)
- generic interfaces (C, Python, Fortran, Julia) : MGIS[1]

reduce code burden and source of errors (different conventions/code)
syntax close to the mathematical constitutive equations

[1] MFrontGenericInterfaceSupport project

A simple example: Norton viscoplasticity



```
@DSL Implicit;
@Behaviour Norton;
@Brick StandardElasticity;

@MaterialProperty stress E;
E.setGlossaryName("YoungModulus");
@MaterialProperty real ν, A, nn;
ν.setGlossaryName("PoissonRatio");
A.setEntryName("NortonCoefficient");
nn.setEntryName("NortonExponent");

@StateVariable real    p;
p.setGlossaryName("EquivalentViscoplasticStrain");

@Integrator{
  constexpr const auto Mᵉ = Stensor4::M();
  const auto μ = computeMu(E, ν);
  const auto σᵉ = sigmaeq(σ);
  const auto iσᵉ = 1 / (max(σᵉ, real(1.e-12) ⋅ E));
  const auto vᵖ = A ⋅ pow(σᵉ, nn);
  const auto ∂vᵖ/∂σᵉ = nn ⋅ vᵖ ⋅ iσᵉ;
  const auto n = 3 ⋅ deviator(σ) ⋅ (iσᵉ / 2);
  // Implicit system
  fεᵉˡ += Δp ⋅ n;
  fp -= vᵖ ⋅ Δt;
  // jacobian
  ∂fεᵉˡ/∂Δεᵉˡ += 2 ⋅ μ ⋅ θ ⋅ dp ⋅ iσᵉ ⋅ (Mᵉ - (n ⊗ n));
  ∂fεᵉˡ/∂Δp = n;
  ∂fp/∂Δεᵉˡ = -2 ⋅ μ ⋅ θ ⋅ ∂vᵖ/∂σᵉ ⋅ Δt ⋅ n;
} // end of @Integrator
```

Implicit system:

$$f_{\varepsilon^e} = \Delta\varepsilon^e - \Delta\varepsilon + \Delta p \boldsymbol{n} = 0$$
$$f_p = \Delta p - A(\sigma_{eq})^n = 0$$

Jacobian:

$$\frac{\partial f_{\varepsilon^e}}{\partial \Delta\varepsilon^e} = \mathbb{I} + \frac{2\mu\theta\Delta p}{\sigma_{eq}}(\mathbb{M} - \boldsymbol{n}\otimes\boldsymbol{n})$$
$$\frac{\partial f_{\varepsilon^e}}{\partial \Delta p} = \boldsymbol{n}$$
$$\frac{\partial f_p}{\partial \Delta\varepsilon^e} = -2\mu\theta\Delta t A n(\sigma_{eq})^{n-1}\boldsymbol{n}$$

tangent operator $\mathbb{C}_t = (J^{-1})_{11}\mathbb{C}$ using the inverse jacobian

Unicode support

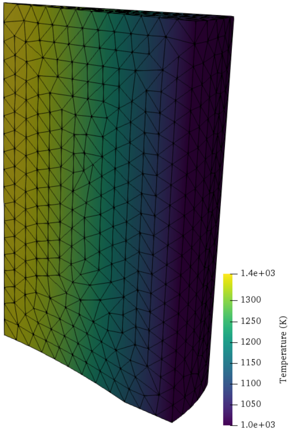
Inside `dolfinx_materials`

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

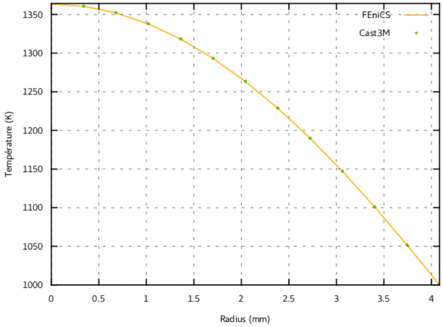
The **only** metadata not provided by **MGIS** is how the gradients (e.g. strain) are expressed as functions of the unknown fields ***u*** (e.g. displacement)
The user is required to provide this link with UFL expressions (**registration**):

```
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)
```

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



Extension to large strains

Quite simple using for instance $\boldsymbol{F} = \boldsymbol{I} + \nabla \boldsymbol{u}$ and $\boldsymbol{P}$ (PK1 stress):
Residual is:

$$\int_\Omega \boldsymbol{P}(\boldsymbol{F}) : \nabla \boldsymbol{v} \, \mathrm{d}\Omega - W_{\text{ext}}(\boldsymbol{v}) = 0 \quad \forall \boldsymbol{v} \in V$$

```
material = MFrontMaterial("src/libBehaviour.so", "Ogden")

qmap = QuadratureMap(domain, deg_quad, material)
qmap.register_gradient("DeformationGradient", F(u))
P = qmap.fluxes["FirstPiolaKirchhoffStress"]
Res = ufl.dot(P, dF(v)) * qmap.dx - ufl.inner(f, v) * dx
```

Extension to large strains

Quite simple using for instance $\boldsymbol{F} = \boldsymbol{I} + \nabla \boldsymbol{u}$ and $\boldsymbol{P}$ (PK1 stress):
Residual is:

$$\int_\Omega \boldsymbol{P}(\boldsymbol{F}) : \nabla \boldsymbol{v} \,\mathrm{d}\Omega - W_{\text{ext}}(\boldsymbol{v}) = 0 \quad \forall \boldsymbol{v} \in V$$

```
material = MFrontMaterial("src/libBehaviour.so", "Ogden")

qmap = QuadratureMap(domain, deg_quad, material)
qmap.register_gradient("DeformationGradient", F(u))
P = qmap.fluxes["FirstPiolaKirchhoffStress"]
Res = ufl.dot(P, dF(v)) * qmap.dx - ufl.inner(f, v) * dx
```

Consistent tangent bilinear form is:

$$a(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \nabla \boldsymbol{u} : \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}} : \nabla \boldsymbol{v} \,\mathrm{d}\Omega$$

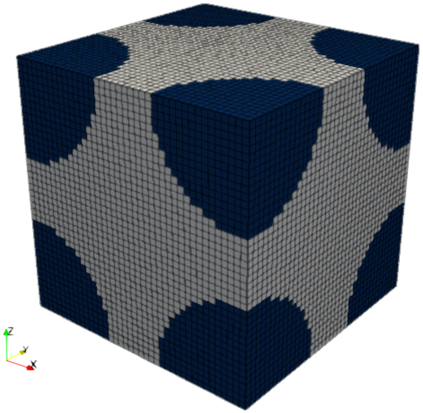
a MFront behaviour usually returns $\mathbb{C}_t = \partial \boldsymbol{\sigma} / \partial \boldsymbol{F}$, MGIS provides conversion methods (**extremely useful**!):

```
bopts = mgis_bv.FiniteStrainBehaviourOptions()
bopts.stress_measure = mgis_bv.FiniteStrainBehaviourOptionsStressMeasure.PK1
bopts.tangent_operator = mgis_bv.FiniteStrainBehaviourOptionsTangentOperator.DPK1_DF
```

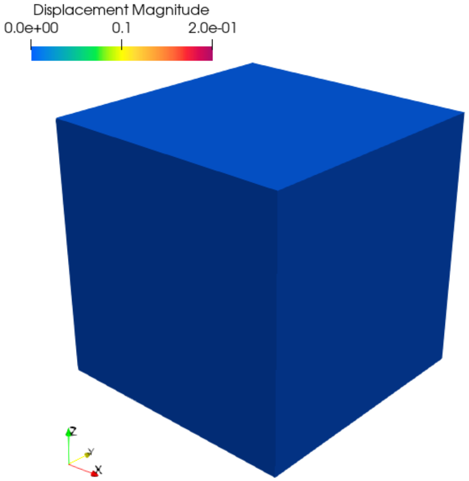
Ogden hyperelasticity

$$\psi(\boldsymbol{F}) = \frac{1}{2}K(J-1)^2 + \sum_{i=1}^{N} \frac{\mu_i}{\alpha_i}\left(\overline{\lambda}_1^{\alpha_i} + \overline{\lambda}_2^{\alpha_i} + \overline{\lambda}_3^{\alpha_i}\right)$$

where $\overline{\lambda}_j$ are eigenvalues of $\overline{C} = J^{-2/3}\boldsymbol{F}^\mathsf{T}\boldsymbol{F}$



Ogden hyperelasticity



16 CPUs: **Linear solves** (x77) = 283.6 s, **Constitutive update** (x98): 20.93 s

Return mapping using convex optimization

For **elasto-plastic problems**, constitutive update is equivalent to **projection of elastic predictor onto the yield surface**:

$$\begin{array}{ll} \min\limits_{\boldsymbol{\sigma}} & \dfrac{1}{2}(\boldsymbol{\sigma}-\boldsymbol{\sigma}_{\text{el}}):\mathbb{C}^{-1}:(\boldsymbol{\sigma}-\boldsymbol{\sigma}_{\text{el}}) \\ \text{s.t.} & f(\boldsymbol{\sigma}) \leq 0 \end{array}$$

local Newton methods may fail for **highly non-smooth surfaces** (Mohr-Coulomb, Rankine, multi-surface)

can use **convex optimization solvers** in such cases (e.g. `cvxpy`)

Return mapping using convex optimization

For **elasto-plastic problems**, constitutive update is equivalent to **projection of elastic predictor onto the yield surface**:

$$\begin{array}{ll} \min_{\boldsymbol{\sigma}} & \frac{1}{2}(\boldsymbol{\sigma}-\boldsymbol{\sigma}_{\text{el}}):\mathbb{C}^{-1}:(\boldsymbol{\sigma}-\boldsymbol{\sigma}_{\text{el}}) \\ \text{s.t.} & f(\boldsymbol{\sigma}) \leq 0 \end{array}$$

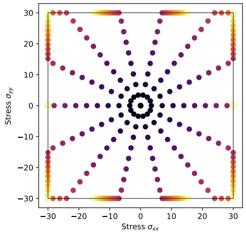
local Newton methods may fail for **highly non-smooth surfaces** (Mohr-Coulomb, Rankine, multi-surface)
can use **convex optimization solvers** in such cases (e.g. `cvxpy`)

e.g. **Rankine**: $f(\boldsymbol{\sigma}) \leq 0 \Leftrightarrow \begin{cases} \max \sigma_I \leq f_t \\ \min \sigma_I \geq -f_c \end{cases}$

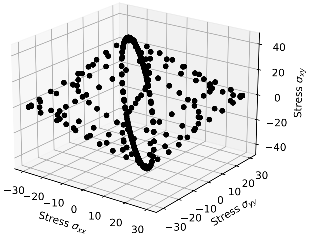
```
import cvxpy as cp
...
def set_cvxpy_model(self):
    self.sig = cp.Variable((3,))
    self.sig_el = cp.Parameter((3,))
    obj = 0.5 * cp.quad_form(self.sig - self.sig_el, np.linalg.inv(C))
    Sig = cp.bmat([[self.sig[0], self.sig[2] / np.sqrt(2)],
                   [self.sig[2] / np.sqrt(2), self.sig[1]]])
    cons = [cp.lambda_max(Sig) <= self.ft,
            cp.lambda_min(Sig) >= -self.fc]
    self.prob = cp.Problem(cp.Minimize(obj), cons)
```

Rankine yield surface

```python
def constitutive_update(self, eps, state):
    eps_old = state["Strain"]
    deps = eps - eps_old
    sig_old = state["Stress"]

    self.sig_el.value = sig_old + self.elastic_model.C @ deps
    self.prob.solve(solver=cp.MOSEK, verbose=False)

    state["Strain"] = eps
    state["Stress"] = self.sig.value
    return sig, self.elastic_model.C
```



(a) In-plane load cases

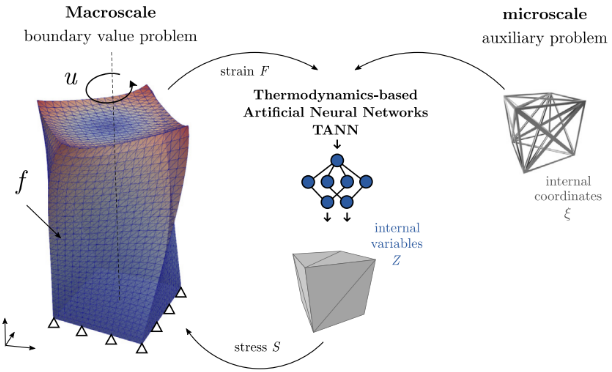


(b) 3D yield surface

Thermodynamics-based Artificial Neural Networks (TANN)

[Masi et al., 2019; Masi and Stefanou, 2022]

NN for the constitutive modeling of materials with **inelastic and complex microstructure**, complying with **thermodynamics requirements**



Multiscale FEM x TANN [Masi and Stefanou, 2022]

constitutive relation of RVE is **trained** on various load paths at the **microlevel**
constitutive relation at the **macrolevel** is **infered** from the trained model

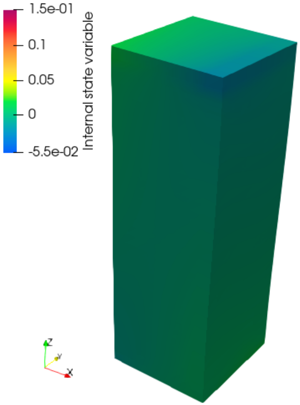
FEM x TANN in dolfinx_materials

```python
import tensorflow as tf
import numpy as np

class TannMaterial(Material):
    def __init__(self, ANN_filename, nb_isv):
        self.model = tf.saved_model.load(ANN_filename)
        self.nb_isv = nb_isv

    @property
    def internal_state_variables(self):
        return {"ivars": self.nb_isv, "free_energy": 1, "dissipation": 1}

    def constitutive_update(self, eps, state):
        state_vars = np.concatenate((state["Strain"], state["Stress"], state["ivars"]))
        deps = eps - state["Strain"]
        inputs = np.concatenate((state_vars, deps))
        stress, svars, Ctang = self.model(inputs, training=False)
```

tangent operator is computed via **NN automatic differentiation**

FEM x TANN in `dolfinx_materials`

Underlying microstructure = 3D truss



Conclusions and Outlook

Project available at:

`https://gitlab.enpc.fr/navier-fenics/dolfinx_materials`

Applications

- **Multiscale** computations: full-field (FE/FFT) or mean-field (micromechanics) models
- **Data-driven** models: projection onto a data manifold
- **Micromorphic regularizations** of softening plasticity models

Technical aspects

- similar to `ExternalOperator` in UFL (current discussions with Jack Hale)
- `JAX/autograd` for automatic differentiation of Python implementations
- `cvxpy` derivatives currently limited to a specific solver and of weak accuracy (Quasi-Newton methods ?)
- constitutive relation to be evaluated **inside the assembly loop**

Conclusions and Outlook

Project available at:

`https://gitlab.enpc.fr/navier-fenics/dolfinx_materials`

Applications

- **Multiscale** computations: full-field (FE/FFT) or mean-field (micromechanics) models
- **Data-driven** models: projection onto a data manifold
- **Micromorphic regularizations** of softening plasticity models

Technical aspects

- similar to `ExternalOperator` in UFL (current discussions with Jack Hale)
- `JAX/autograd` for automatic differentiation of Python implementations
- `cvxpy` derivatives currently limited to a specific solver and of weak accuracy (Quasi-Newton methods ?)
- constitutive relation to be evaluated **inside the assembly loop**

Thank you for your attention !