Binding `MFront` with `FEniCS` for automated formulation and resolution of non-linear behaviours

Jeremy Bleyer, Thomas Helfer

Laboratoire Navier, UMR 8205
Ecole des Ponts ParisTech-Univ Gustave Eiffel-CNRS

6th `MFront` User Meeting

November, 25th 2020

Outline

Nonlinear problems - Hyperelasticity

elastic potential + automatic differentiation with **UFL** operators

```
# Create mesh and define function space
mesh = UnitCubeMesh(24, 16, 16)
V = VectorFunctionSpace(mesh, "Lagrange", 1)

# Define functions
u  = Function(V)                # Displacement from previous iteration
v  = TestFunction(V)            # Test function
du = TrialFunction(V)           # Incremental displacement
dim = len(u)

F = Identity(dim) + grad(u)     # Deformation gradient
C = F.T*F
Ic, J = tr(C), det(F)           # Invariants of deformation tensors

# Elasticity parameters
E, nu = 10.0, 0.3
mu, lmbda = Constant(E/(2*(1 + nu))), Constant(E*nu/((1 + nu)*(1 - 2*nu)))
# potential (compressible neo-Hookean model)
psi = (mu/2)*(Ic - 3) - mu*ln(J) + (lmbda/2)*(ln(J))**2

# Total potential energy
f = Constant((0.0, -1.0, 0.0))
Pi = psi*dx - dot(f, u)*dx
```

Nonlinear problems - Hyperelasticity

```
# Compute first variation of Pi (directional derivative)
R = derivative(Pi, u, v)

# Compute Jacobian of R
J = derivative(R, u, du)

# Solve variational problem
solve(R == 0, u, bc, J=J)
```

we can specify more precisely which solver and parameters to use

Nonlinear problems - Hyperelasticity

```
# Compute first variation of Pi (directional derivative)
R = derivative(Pi, u, v)

# Compute Jacobian of R
J = derivative(R, u, du)

# Solve variational problem
solve(R == 0, u, bc, J=J)
```

we can specify more precisely which solver and parameters to use

- Newton method
- PETSc SNES solver : line search, trust region
- PETSC TAO solver : bound-constrained minimization

we can choose which linear solver and preconditioner to use for the iterative process

also possible to **formulate yourself** a Newton method at the PDE level

Outline

Generic behaviours

Mechanics:

$$\Delta\varepsilon, \boldsymbol{\sigma}_n, \boldsymbol{Y}_n \to \boxed{\text{MFront}} \to \boldsymbol{\sigma}_{n+1}, \boldsymbol{Y}_{n+1}, \frac{\partial \boldsymbol{\sigma}}{\partial \Delta\varepsilon}$$

Generalized behaviours:

$$(\Delta\boldsymbol{g}^1, \dots \Delta\boldsymbol{g}^p)_n, (\boldsymbol{\sigma}^1, \dots, \boldsymbol{\sigma}^p)_n, \boldsymbol{Y}_n \to \boxed{\text{MFront}} \to (\boldsymbol{\sigma}^1, \dots, \boldsymbol{\sigma}^p)_{n+1}, \boldsymbol{Y}_{n+1}, \frac{\partial \boldsymbol{\sigma}^j}{\partial \Delta\boldsymbol{g}^k}$$

$\boldsymbol{g}^j$ are **gradients** (temp. gradient, strain, etc.) depending on the FE unknowns $\boldsymbol{u}$
$\boldsymbol{\sigma}^j$ are associated **fluxes** or **thermodynamic forces** (heat flux, stress, etc.)
$\dfrac{\partial \boldsymbol{\sigma}^j}{\partial \Delta\boldsymbol{g}^k}$ are **tangent blocks**
Work of internal forces density:

$$\delta w_{\text{int}} = \sum_{i=1}^{p} \boldsymbol{\sigma}^i \cdot \delta\boldsymbol{g}^i(\boldsymbol{v})$$

mgis.fenics Python package - Generalities

relies on the MGIS Python interface for loading a MFront library, calling the behaviour integration and giving access to fluxes, state variables and tangent operators

mgis.fenics Python package - Generalities

relies on the MGIS Python interface for loading a MFront library, calling the behaviour integration and giving access to fluxes, state variables and tangent operators

General idea

Inside a custom `NonlinearProblem`, define $\boldsymbol{\sigma}^i$ on a `Quadrature` space and the generalized residual:

$$F(\boldsymbol{v}) = \sum_{i=1}^{p} \int_\Omega \boldsymbol{\sigma}^i(\boldsymbol{u}) \cdot \delta \boldsymbol{g}^i(\boldsymbol{v}) \,\mathrm{dx} - L(\boldsymbol{v}) = 0 \quad \forall \boldsymbol{v} \in V$$

mgis.fenics Python package - Generalities

relies on the MGIS Python interface for loading a MFront library, calling the behaviour integration and giving access to fluxes, state variables and tangent operators

General idea

Inside a custom `NonlinearProblem`, define $\boldsymbol{\sigma}^i$ on a `Quadrature` space and the generalized residual:

$$F(\boldsymbol{v}) = \sum_{i=1}^{p} \int_\Omega \boldsymbol{\sigma}^i(\boldsymbol{u}) \cdot \delta \boldsymbol{g}^i(\boldsymbol{v}) \,\mathrm{dx} - L(\boldsymbol{v}) = 0 \quad \forall \boldsymbol{v} \in V$$

MGIS gives metadata to know on which blocks $\mathcal{B}(i)$ of gradients each flux $\boldsymbol{\sigma}_i$ depends:

$$a_{\text{tangent}}(\boldsymbol{u}, \boldsymbol{v}) = \sum_{i=1}^{p} \sum_{j \in \mathcal{B}(i)} \int_\Omega \delta \boldsymbol{g}^i(\boldsymbol{v}) \cdot \mathbb{T}_{\boldsymbol{g}^j}^{\boldsymbol{\sigma}^i} \cdot \delta \boldsymbol{g}^j(\boldsymbol{u}) \,\mathrm{dx}$$

with each tangent block $\mathbb{T}_{\boldsymbol{g}^j}^{\boldsymbol{\sigma}^i} = \dfrac{\partial \boldsymbol{\sigma}^i}{\partial \boldsymbol{g}^j}$ represented on a tensorial `Quadrature` space

mgis.fenics Python package - Registration concept

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)

mgis.fenics Python package - Registration concept

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 = mf.MFrontNonlinearMaterial("src/libBehaviour.so",
               "IsotropicLinearHardeningPlasticity",
               hypothesis="plane_strain",
               material_properties=mat_prop)

u = Function(V)
problem = mf.MFrontNonlinearProblem(u, material,
                    quadrature_degree=2, bcs=bc)
problem.register_gradient("Strain", sym(grad(u)))
```

mgis.fenics Python package - Registration concept

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 = mf.MFrontNonlinearMaterial("src/libBehaviour.so",
               "IsotropicLinearHardeningPlasticity",
               hypothesis="plane_strain",
               material_properties=mat_prop)

u = Function(V)
problem = mf.MFrontNonlinearProblem(u, material,
                    quadrature_degree=2, bcs=bc)
problem.register_gradient("Strain", sym(grad(u)))
```

can be done automatically for predefined keywords e.g. `"Strain"`, `"TemperatureGradient"`, etc.

mgis.fenics Python package - Overview

UFL `eps = sym(grad(u))`

registration `{"Strain": eps}`

array of values at integ. points `project(eps, Q)` → **MFront** → `sig_values` `Ct_values`

```
sig[:] = sig_values
Ct[:]  = Ct_values
```

Quadrature functions

```
F_int = inner(sig, eps_)*dx
```

UFL symbolic differentiation

```
a_tang = inner(inner(Ct, deps), eps_)*dx
```

```
assemble(F_int)
assemble(a_tang)
```
→ `solve`

nonlinear solver (Newton, SNES)

mgis.fenics Python package - Limitations

Pros

- Any mechanical behaviour can be solved easily.
- Can modify the residual form to account for time-dependent problems → automatic differentiation is preserved using MGIS metadata.
- Non-linear solvers can be chosen independently.
- Works in parallel.

mgis.fenics Python package - Limitations

Pros

- Any mechanical behaviour can be solved easily.
- Can modify the residual form to account for time-dependent problems $\rightarrow$ automatic differentiation is preserved using MGIS metadata.
- Non-linear solvers can be chosen independently.
- Works in parallel.

Cons

- Constitutive update (call to MFront) is performed outside the assembler ($\Rightarrow$ one extra loop over quadrature points) and all tangent blocks $\mathbb{T}$ at all quadrature points must be stored in memory.
- Multi-material support is limited to spatially varying material properties.
- Memory transfers between FEniCS and MGIS objects have not been optimized.

$\Rightarrow$ to be improved with **FEniCS-X**

Outline

Stationary non-linear heat transfer

Non-linear Fourier law (UO_2): $\boldsymbol{j}(\nabla T, T) = -k(T)\nabla T$ with $k(T) = \dfrac{1}{A + BT}$ (✓)

$$a_{\mathrm{tangent}}(\widehat{T}, T^*) = \int_\Omega \nabla \widehat{T} \cdot \left(\frac{\partial \boldsymbol{j}}{\partial \boldsymbol{g}} \cdot \nabla T^* + \frac{\partial \boldsymbol{j}}{\partial T} \cdot T^* \right) \mathrm{dx}$$

Stationary non-linear heat transfer

Non-linear Fourier law (UO_2): $\boldsymbol{j}(\nabla T, T) = -k(T)\nabla T$ with $k(T) = \dfrac{1}{A + BT}$ (✓)

$$a_{\text{tangent}}(\widehat{T}, T^*) = \int_\Omega \nabla \widehat{T} \cdot \left(\frac{\partial \boldsymbol{j}}{\partial \boldsymbol{g}} \cdot \nabla T^* + \frac{\partial \boldsymbol{j}}{\partial T} \cdot T^* \right) \mathrm{dx}$$

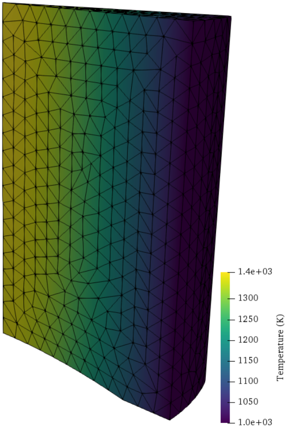
```
@DSL DefaultGenericBehaviour;
@Behaviour StationaryHeatTransfer;

@Gradient TemperatureGradient ∇T;
∇T.setGlossaryName("TemperatureGradient");

@Flux HeatFlux j;
j.setGlossaryName("HeatFlux");

@Integrator{
  // temperature at the end of the time step
  const auto T_ = T + ΔT;
  // thermal conductivity
  k = 1 / (A + B ⋅ T_);
  // heat flux
  j = -k ⋅ (∇T + Δ∇T);
} // end of @Integrator

@AdditionalTangentOperatorBlock ∂j/∂ΔT;
@TangentOperator {
  ∂j/∂Δ∇T = -k ⋅ tmatrix<N, N, real>::Id();
  ∂j/∂ΔT  =  B ⋅ k ⋅ k ⋅ (∇T + Δ∇T);
} // end of @TangentOperator
```

```
print(["d{}_d{}".format(*t) for t in material.
      get_tangent_block_names()])
```

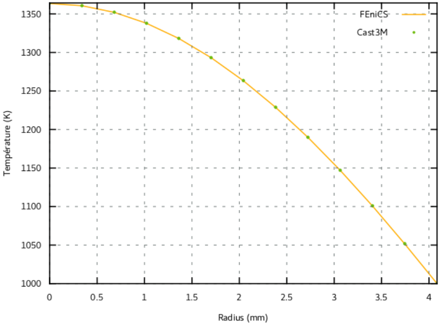
```
['dHeatFlux_dTemperatureGradient',
 'dHeatFlux_dTemperature']
```

```
V = FunctionSpace(mesh, "CG", 1)
T = Function(V, name="Temperature")

problem = mf.MFrontNonlinearProblem(T, material,
      quadrature_degree=2, bcs=bc)
problem.set_loading(-r*T*dx)

problem.register_gradient("TemperatureGradient",
      grad(T))

problem.solve(T.vector())
```

Stationary non-linear heat transfer



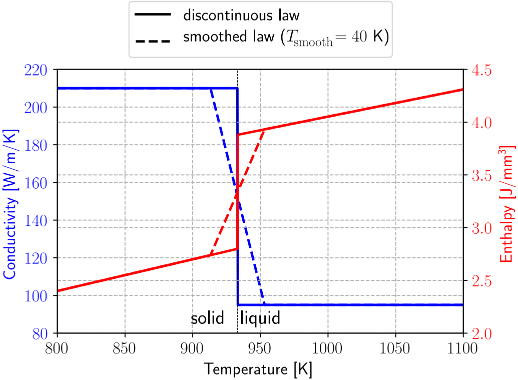
Mesh type	quad_deg	MFront	pure FEniCS
coarse	2	1.2 s	0.8 s
coarse	5	2.2 s	1.0 s
fine	2	62.8 s	58.4 s
fine	5	77.0 s	66.3 s



Transient heat equation with phase change

Transient heat equation with enthalpy formulation:

$$\frac{\partial h}{\partial t} = r - \operatorname{div} \boldsymbol{j}$$
$$\boldsymbol{j}(T, \nabla T) = -k(T)\nabla T$$



$$h(T) = \int_{T_0}^{T} \rho C_p dT = \begin{cases} c_s T & \text{if } T < T_m \\ c_l(T - T_m) + c_s T_m + \Delta h_{s/l} & \text{if } T > T_m \end{cases}$$

Smoothed version $T_{\text{smooth}} = 0.1$ K

Transient heat equation with phase change

Time integration with θ-scheme: $\star_{n+\theta} = \theta \star_{n+1} + (1-\theta)\star_n$

$$\left.\frac{\partial h}{\partial t}\right|_{t=t_{n+\theta}} \approx \frac{h_{n+1} - h_n}{\Delta t} = r_{n+\theta} - \operatorname{div} \boldsymbol{j}_{n+\theta}$$

Transient heat equation with phase change

Time integration with θ-scheme: $\star_{n+\theta} = \theta \star_{n+1} + (1-\theta)\star_n$

$$\left.\frac{\partial h}{\partial t}\right|_{t=t_{n+\theta}} \approx \frac{h_{n+1} - h_n}{\Delta t} = r_{n+\theta} - \operatorname{div} \boldsymbol{j}_{n+\theta}$$

$$F(\widehat{T}) = \int_\Omega \left((h_{n+1}(T) - h_n)\,\widehat{T} - \Delta t(\theta \boldsymbol{j}_{n+1}(T, \nabla T) + (1-\theta)\boldsymbol{j}_n) \cdot \nabla \widehat{T} \right) \,\mathrm{dx} = 0$$

Transient heat equation with phase change

Time integration with θ-scheme: $\star_{n+\theta} = \theta \star_{n+1} + (1-\theta)\star_n$

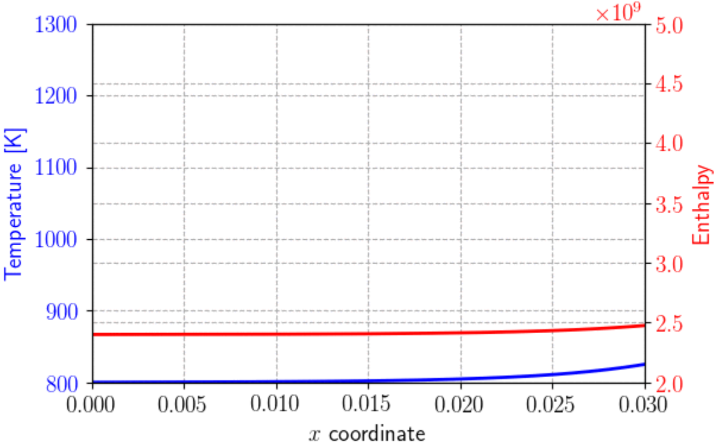
$$\left.\frac{\partial h}{\partial t}\right|_{t=t_{n+\theta}} \approx \frac{h_{n+1} - h_n}{\Delta t} = r_{n+\theta} - \operatorname{div} \boldsymbol{j}_{n+\theta}$$

$$F(\widehat{T}) = \int_\Omega \left((h_{n+1}(T) - h_n)\widehat{T} - \Delta t(\theta \boldsymbol{j}_{n+1}(T, \nabla T) + (1-\theta)\boldsymbol{j}_n) \cdot \nabla \widehat{T} \right) \, \mathrm{dx} = 0$$

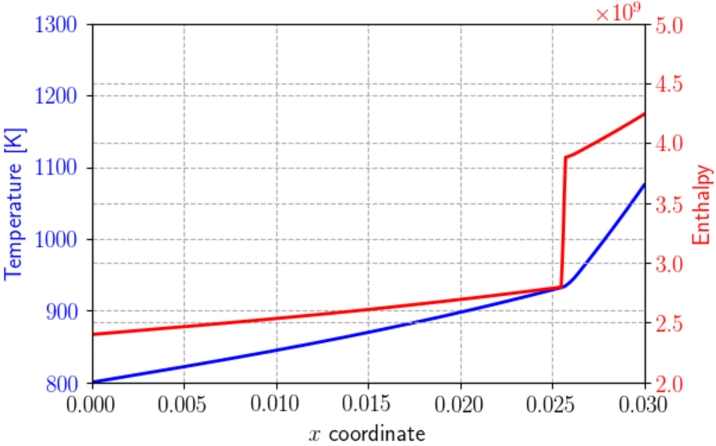
```
@DSL DefaultGenericBehaviour;
@Behaviour HeatTransferPhaseChange;

@Gradient TemperatureGradient ∇T;
∇T.setGlossaryName("TemperatureGradient");

@Flux HeatFlux j;
j.setGlossaryName("HeatFlux");

@StateVariable real h;
h.setEntryName("Enthalpy"); //per unit of volume

@AdditionalTangentOperatorBlock ∂j/∂ΔT;
@AdditionalTangentOperatorBlock ∂h/∂ΔT;
```

```
h = problem.get_state_variable("Enthalpy")
j = problem.get_flux("HeatFlux")
problem.initialize()

j_old = j.copy(deepcopy=True)
h_old = h.copy(deepcopy=True)

j_theta = theta*j + (1-theta)*j_old
problem.residual = (T_*(h - h_old)-dt*dot(
    grad(T_), j_theta))*problem.dx
problem.compute_tangent_form()
```

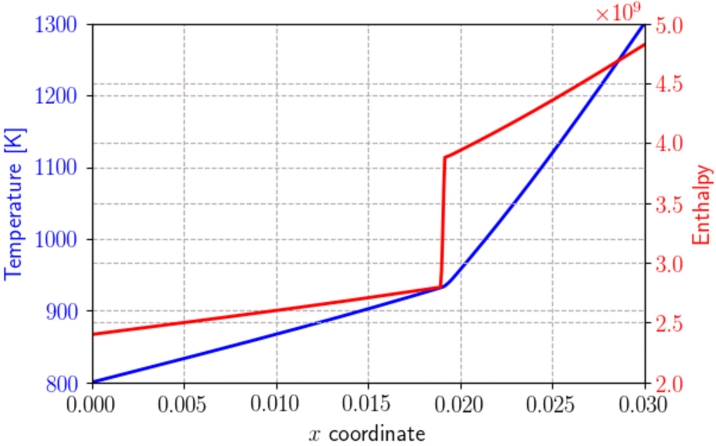
Transient heat equation with phase change



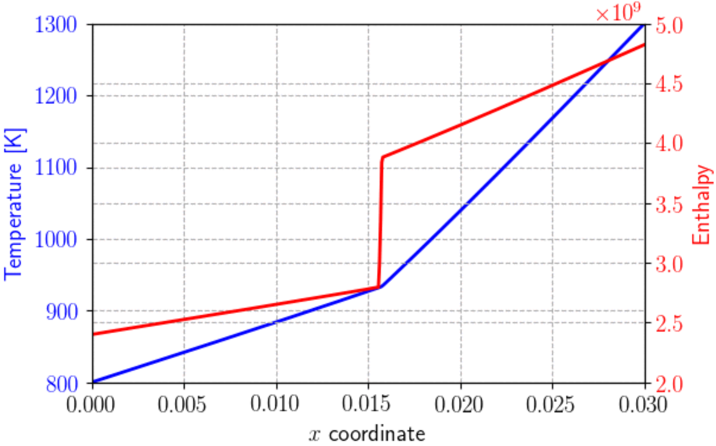
Transient heat equation with phase change



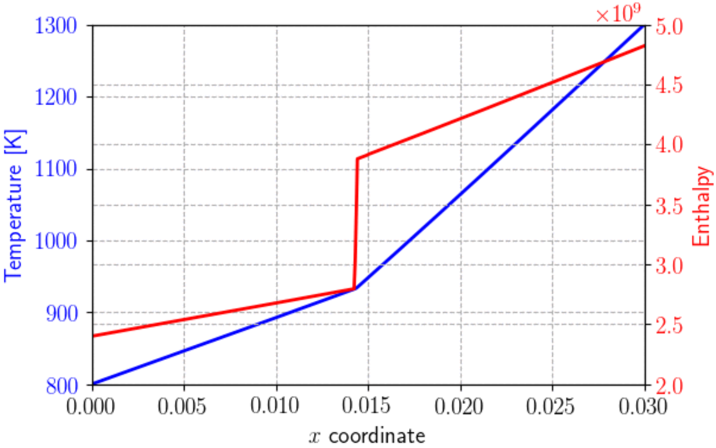
Transient heat equation with phase change



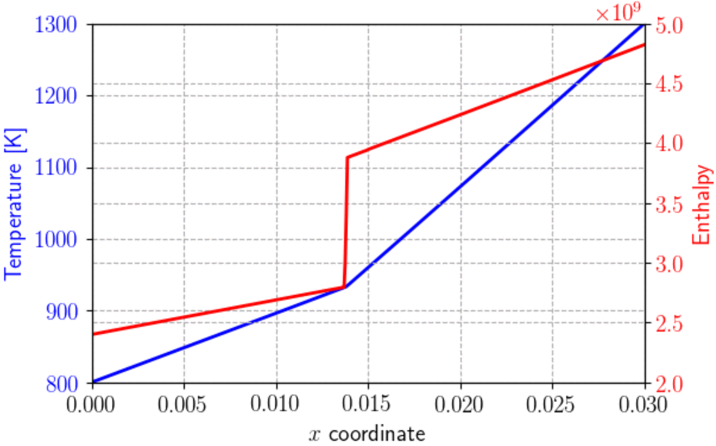
Transient heat equation with phase change



Transient heat equation with phase change



Transient heat equation with phase change



Extension to large strains

Quite simple using for instance $\boldsymbol{F}$ (*gradient*) and PK1 stress $\boldsymbol{P}$ (*flux*):
Residual is:

$$F(\boldsymbol{v}) = \int_\Omega \boldsymbol{P} : \delta\boldsymbol{F}(\boldsymbol{v}) \text{ dx} - W_{ext}(\boldsymbol{v}) = \int_\Omega \boldsymbol{P} : \nabla\boldsymbol{v} \text{ dx} - W_{ext}(\boldsymbol{v})$$

$\boldsymbol{F}$ and $\boldsymbol{P}$ are non-symmetric 2nd-order tensors

Extension to large strains

Quite simple using for instance $\boldsymbol{F}$ (*gradient*) and PK1 stress $\boldsymbol{P}$ (*flux*):
Residual is:

$$F(\boldsymbol{v}) = \int_\Omega \boldsymbol{P} : \delta\boldsymbol{F}(\boldsymbol{v}) \,\mathrm{dx} - W_{ext}(\boldsymbol{v}) = \int_\Omega \boldsymbol{P} : \nabla \boldsymbol{v} \,\mathrm{dx} - W_{ext}(\boldsymbol{v})$$

$\boldsymbol{F}$ and $\boldsymbol{P}$ are non-symmetric 2nd-order tensors
Consistent tangent bilinear form is:

$$a_{\text{tangent}}(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \nabla \boldsymbol{u} : \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}} : \nabla \boldsymbol{v} dx$$

3 possible tangent operators $\dfrac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{F}}, \dfrac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}}, \dfrac{\partial \boldsymbol{S}}{\partial \boldsymbol{E}_{GL}}$, obtained from the behaviour tangent operator using MFront conversion methods (extremely useful!)

Extension to large strains

Quite simple using for instance $\boldsymbol{F}$ (*gradient*) and PK1 stress $\boldsymbol{P}$ (*flux*):
Residual is:

$$F(\boldsymbol{v}) = \int_\Omega \boldsymbol{P} : \delta\boldsymbol{F}(\boldsymbol{v}) \,\mathrm{dx} - W_{ext}(\boldsymbol{v}) = \int_\Omega \boldsymbol{P} : \nabla\boldsymbol{v} \,\mathrm{dx} - W_{ext}(\boldsymbol{v})$$

$\boldsymbol{F}$ and $\boldsymbol{P}$ are non-symmetric 2nd-order tensors
Consistent tangent bilinear form is:

$$a_{\text{tangent}}(\boldsymbol{u}, \boldsymbol{v}) = \int_\Omega \nabla\boldsymbol{u} : \frac{\partial\boldsymbol{P}}{\partial\boldsymbol{F}} : \nabla\boldsymbol{v} dx$$

3 possible tangent operators $\dfrac{\partial\boldsymbol{\sigma}}{\partial\boldsymbol{F}}, \dfrac{\partial\boldsymbol{P}}{\partial\boldsymbol{F}}, \dfrac{\partial\boldsymbol{S}}{\partial\boldsymbol{E}_{GL}}$, obtained from the behaviour tangent operator using MFront conversion methods (extremely useful!)

Logarithmic strain plasticity

- Hencky strain measure $\boldsymbol{H} = \dfrac{1}{2}\log(\boldsymbol{F}^{\mathrm{T}} \cdot \boldsymbol{F})$ ✗
- use a small strain constitutive relation on $\boldsymbol{H} \Rightarrow \boldsymbol{H} = \boldsymbol{H}^e + \boldsymbol{H}^p$

$$\boldsymbol{F} \longrightarrow \boldsymbol{H} \longrightarrow \boxed{\text{small strain law}} \longrightarrow \boldsymbol{T} \longrightarrow \boldsymbol{\sigma}$$

Logarithmic strain plasticity

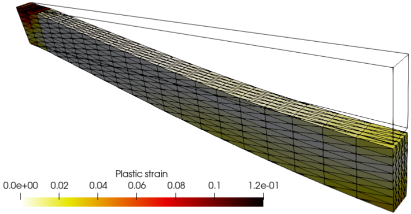
Use of the StandardElastoViscoPlasticity brick

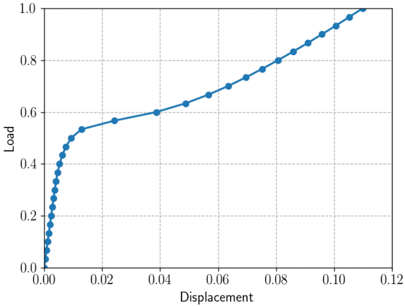
```
@DSL Implicit;

@Behaviour LogarithmicStrainPlasticity;

@StrainMeasure Hencky;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {
        young_modulus : 210e9,
        poisson_ratio : 0.3
        },
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "Linear" {H : 500e6,
                                    R0 : 250e6}
  }
};
```

```
material = mf.MFrontNonlinearMaterial("./src/
     libBehaviour.so", "
     LogarithmicStrainPlasticity")
problem = mf.MFrontNonlinearProblem(u,
     material, bcs=bc)
problem.set_loading(dot(selfweight, u)*dx)

prm = problem.solver.parameters
prm["absolute_tolerance"] = 1e-6
prm["relative_tolerance"] = 1e-6
prm["linear_solver"] = "mumps"

for (i, t) in enumerate(load_steps[1:]):
    selfweight.t = t
    problem.solve(u.vector())
```

```
Automatic registration of 'DeformationGradient'
as I + (grad(Displacement))
```

Logarithmic strain plasticity





Phase-field approach to brittle fracture

Bourdin/Francfort/Marigo **variational phase-field approach**:

$$\boldsymbol{u}(t), d(t) = \underset{\boldsymbol{u},d}{\arg\min}\, \mathcal{E}_{pot}(\boldsymbol{u}, d) + \mathcal{E}_f(\boldsymbol{u}, d)$$

$$\mathcal{E}_{pot}(\boldsymbol{u}, d) = \int_\Omega (1-d)^2\psi^+(\boldsymbol{\varepsilon}) + \psi^-(\boldsymbol{\varepsilon}) \text{ dx} - W_{ext}(\boldsymbol{u})$$

$$\mathcal{E}_f(d) = \dfrac{G_c}{c_w}\int_\Omega \left(\dfrac{w(d)}{\ell_0} + \ell_0\|\nabla d\|^2\right) \text{ dx}$$

Phase-field approach to brittle fracture

Bourdin/Francfort/Marigo **variational phase-field approach**:

$$\boldsymbol{u}(t), d(t) = \underset{\boldsymbol{u},d}{\arg\min}\, \mathcal{E}_{pot}(\boldsymbol{u}, d) + \mathcal{E}_f(\boldsymbol{u}, d)$$

$$\mathcal{E}_{pot}(\boldsymbol{u}, d) = \int_\Omega (1-d)^2\psi^+(\boldsymbol{\varepsilon}) + \psi^-(\boldsymbol{\varepsilon}) \,\mathrm{dx} - W_{ext}(\boldsymbol{u})$$

$$\mathcal{E}_f(d) = \frac{G_c}{c_w}\int_\Omega \left(\frac{w(d)}{\ell_0} + \ell_0\|\nabla d\|^2\right) \,\mathrm{dx}$$

tension/compression splittings:

$$\psi^+(\boldsymbol{\varepsilon}) = \frac{1}{2}\kappa\langle \mathrm{tr}(\boldsymbol{\varepsilon})\rangle_+^2 + \mu\boldsymbol{\varepsilon}^{dev} : \boldsymbol{\varepsilon}^{dev} \quad \text{(Amor et al.)} \ \checkmark$$

$$\psi^+(\boldsymbol{\varepsilon}) = \frac{1}{2}\lambda\langle \mathrm{tr}(\boldsymbol{\varepsilon})\rangle_+^2 + \mu\sum_I \langle \varepsilon_I\rangle_+^2 \quad \text{(Miehe et al.)} \ \text{✗}$$

Phase-field approach to brittle fracture

Alternate minimization

$$\boldsymbol{u}^k = \arg\min_{u} \mathcal{E}(\boldsymbol{u}, d^{k-1}) \qquad (u\text{-problem})$$

$$d^k = \arg\min_{d \text{ s.t. } d_{n-\mathbf{1}} \leq d} \mathcal{E}(\boldsymbol{u}^k, d) \qquad (d\text{-problem})$$

Phase-field approach to brittle fracture

Alternate minimization

$$\boldsymbol{u}^k = \arg\min_{u} \mathcal{E}(\boldsymbol{u}, d^{k-1}) \qquad (u\text{-problem})$$

$$d^k = \arg\min_{d \text{ s.t. } d_{n-1} \leq d} \mathcal{E}(\boldsymbol{u}^k, d) \qquad (d\text{-problem})$$

(u-problem) is a **non-linear elasticity** problem with

$\boldsymbol{\sigma}(\boldsymbol{u}, d) = (1-d)^2 \dfrac{\partial \psi^+}{\partial \boldsymbol{\varepsilon}}(\boldsymbol{u}) + \dfrac{\partial \psi^-}{\partial \boldsymbol{\varepsilon}}(\boldsymbol{u})$ and d as an external state variable

Phase-field approach to brittle fracture

Alternate minimization

$$\boldsymbol{u}^k = \arg\min_{u} \mathcal{E}(\boldsymbol{u}, d^{k-1}) \qquad (u\text{-problem})$$

$$d^k = \underset{d \text{ s.t. } d_{n-1}\leq d}{\arg\min} \mathcal{E}(\boldsymbol{u}^k, d) \qquad (d\text{-problem})$$

(u-problem) is a **non-linear elasticity** problem with $\boldsymbol{\sigma}(\boldsymbol{u}, d) = (1-d)^2\dfrac{\partial\psi^+}{\partial\boldsymbol{\varepsilon}}(\boldsymbol{u}) + \dfrac{\partial\psi^-}{\partial\boldsymbol{\varepsilon}}(\boldsymbol{u})$ and d as an external state variable

(d-problem) is **variational inequality** problem with ψ^+ as an external state variable: solved with `PETScTAOSolver`

Phase-field approach to brittle fracture

Alternate minimization

$$\boldsymbol{u}^k = \arg\min_u \mathcal{E}(\boldsymbol{u}, d^{k-1}) \qquad (u\text{-problem})$$

$$d^k = \arg\min_{d \text{ s.t. } d_{n-1} \leq d} \mathcal{E}(\boldsymbol{u}^k, d) \qquad (d\text{-problem})$$

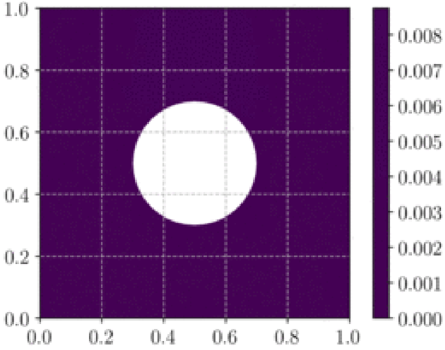
(u-problem) is a **non-linear elasticity** problem with

$\boldsymbol{\sigma}(\boldsymbol{u}, d) = (1-d)^2 \dfrac{\partial \psi^+}{\partial \boldsymbol{\varepsilon}}(\boldsymbol{u}) + \dfrac{\partial \psi^-}{\partial \boldsymbol{\varepsilon}}(\boldsymbol{u})$ and d as an external state variable

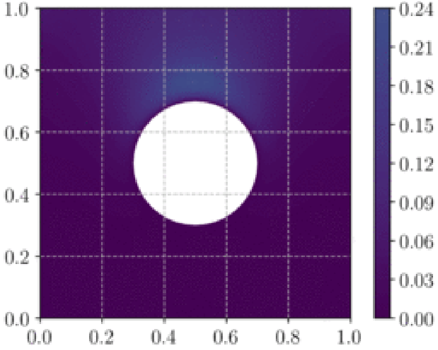
(d-problem) is **variational inequality** problem with ψ^+ as an external state variable: solved with `PETScTAOSolver`

```
problem_u = mf.MFrontNonlinearProblem(u, material_u, bcs=bcu)
problem_u.register_external_state_variable("Damage", d)
psi = problem_u.get_state_variable("PositiveEnergyDensity")

for (i, t) in enumerate(loading[1:]):
    Uimp.t = t
    while res > tol and j < Nitermax:
        problem_u.solve(u.vector()) # Solve displacement u-problem
        problem_d.solve(d.vector()) # Solve damage d-problem
```

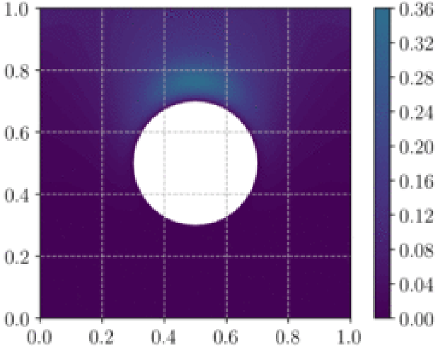
Phase-field approach to brittle fracture



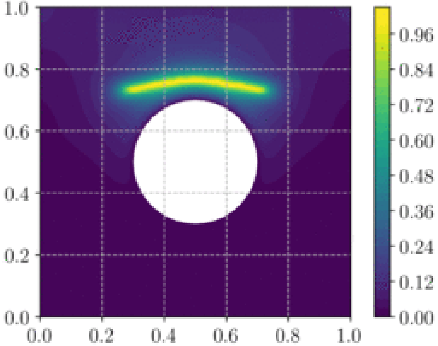
Phase-field approach to brittle fracture



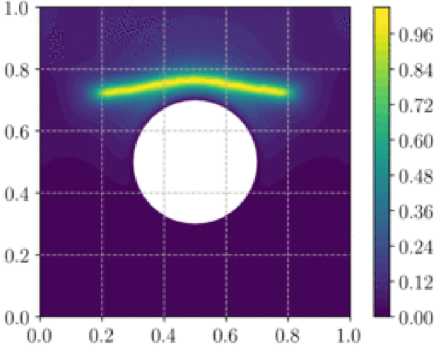
Phase-field approach to brittle fracture



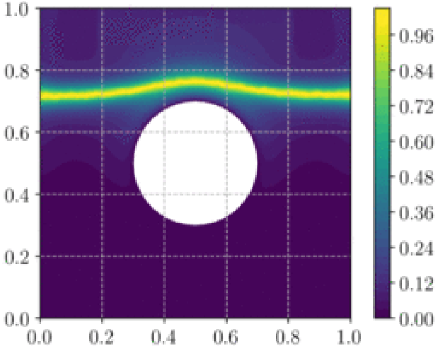
Phase-field approach to brittle fracture



Phase-field approach to brittle fracture



Phase-field approach to brittle fracture



Monolithic thermoelastoplasticity - MFront

Fully coupled thermomechanics ($\varepsilon^{\text{el}} = \varepsilon^{\text{to}} - \varepsilon^{\text{p}} - \varepsilon^{\text{th}}$):

$$\boldsymbol{\sigma} = \lambda\,\mathrm{tr}(\varepsilon^{\text{el}})\boldsymbol{I} + 2\mu\varepsilon^{\text{el}}$$

$$s = \frac{C_\varepsilon}{T^{\text{ref}}}(T - T^{\text{ref}}) + \frac{\kappa}{\rho}\,\mathrm{tr}(\varepsilon^{\text{el}})$$

$$\boldsymbol{j} = -k\nabla T$$

```
@Gradient StrainStensor εᵗᵒ;
εᵗᵒ.setGlossaryName("Strain");

@Flux StressStensor σ;
σ.setGlossaryName("Stress");

@Gradient TemperatureGradient ∇T;
∇T.setGlossaryName("TemperatureGradient");

@Flux HeatFlux j;
j.setGlossaryName("HeatFlux");

@StateVariable StrainStensor eel;
eel.setGlossaryName("ElasticStrain");
@StateVariable strain p;
p.setGlossaryName("EquivalentPlasticStrain");
@StateVariable real s;
s.setEntryName("EntropyPerUnitOfMass");

@TangentOperatorBlocks{∂σ/∂Δεᵗᵒ, ∂σ/∂ΔT, ∂s/∂ΔT, ∂s/∂Δεᵗᵒ, ∂j/∂Δ∇T};
```

```
@Integrator{
  const auto λ = computeLambda(E, ν);
  const auto μ = computeMu(E, ν);
  const auto κ = α ⋅ (2 ⋅ μ + 3 ⋅ λ);
  const auto ε = εᵗᵒ + Δεᵗᵒ;
  const auto εth = α ⋅ (T + ΔT - Tref) ⋅ I₂;

  eel += Δεᵗᵒ - εth;
  const auto se     = 2*μ*deviator(eel);
  const auto seq_e  = sigmaeq(se);
  const auto b      = seq_e-s0-H*p>stress{0};
  if(b){
```

[...]

```
  σ = λ ⋅ trace(eel) ⋅ I₂ + 2 ⋅ μ ⋅ eel ;
  s = Cₑ / Tref ⋅ (T + ΔT - Tref) + (κ / ρ) ⋅ trace(ε);
  j = -k ⋅ (∇T + Δ∇T);
  if (computeTangentOperator_) {
    ∂σ/∂ΔT = -κ ⋅ I₂;
    ∂s/∂ΔT = Cₑ / Tref;
    ∂s/∂Δεᵗᵒ = κ / ρ ⋅ I₂;
    ∂j/∂Δ∇T = -k ⋅ tmatrix<N, N, real>::Id();
  }
}
```

Monolithic thermoelastoplasticity - FEniCS

Quasi-static equilibrium + heat equation:

$$\operatorname{div} \boldsymbol{\sigma} = 0$$

$$\rho T^{\text{ref}} \frac{\partial s}{\partial t} + \operatorname{div} \boldsymbol{j} = 0$$

Monolithic thermoelastoplasticity - FEniCS

Variational formulation and implicit backward Euler:

$$\int_\Omega \boldsymbol{\sigma}_{n+1} : \nabla^s \widehat{\boldsymbol{u}} \,\mathrm{dx} = \int_{\partial\Omega} \boldsymbol{T} \cdot \widehat{\boldsymbol{u}} \,\mathrm{dS} \quad \forall \widehat{\boldsymbol{u}}$$

$$\int_\Omega \left(\rho T^{\mathrm{ref}} \frac{s_{n+1} - s_n}{\Delta t} - \boldsymbol{j}_{n+1}\right) \widehat{T} \,\mathrm{dx} = -\int_{\partial\Omega} q \widehat{T} \,\mathrm{dS} \quad \forall \widehat{T}$$

Monolithic thermoelastoplasticity - FEniCS

Variational formulation and implicit backward Euler:

$$\int_\Omega \boldsymbol{\sigma}_{n+1} : \nabla^s \widehat{\boldsymbol{u}} \,\mathrm{dx} = \int_{\partial\Omega} \boldsymbol{T} \cdot \widehat{\boldsymbol{u}} \,\mathrm{dS} \quad \forall \widehat{\boldsymbol{u}}$$

$$\int_\Omega (\rho T^{\text{ref}} \frac{s_{n+1} - s_n}{\Delta t} - \boldsymbol{j}_{n+1}) \widehat{T} \,\mathrm{dx} = -\int_{\partial\Omega} q \widehat{T} \,\mathrm{dS} \quad \forall \widehat{T}$$

```
Vue = VectorElement("CG", mesh.ufl_cell(), 2)  # displacement finite element
Vte = FiniteElement("CG", mesh.ufl_cell(), 1)  # temperature finite element
V = FunctionSpace(mesh, MixedElement([Vue, Vte]))

v = Function(V)
(u, Theta) = split(v)
problem = mf.MFrontNonlinearProblem(v, material, quadrature_degree=2, bcs=bcs)
problem.register_gradient("Strain", sym(grad(u)))
problem.register_gradient("TemperatureGradient", grad(Theta))
problem.register_external_state_variable("Temperature", Theta + Tref)
```

Monolithic thermoelastoplasticity - FEniCS

Variational formulation and implicit backward Euler:

$$\int_\Omega \boldsymbol{\sigma}_{n+1} : \nabla^s \widehat{\boldsymbol{u}} \, \mathrm{dx} = \int_{\partial\Omega} \boldsymbol{T} \cdot \widehat{\boldsymbol{u}} \, \mathrm{dS} \quad \forall \widehat{\boldsymbol{u}}$$

$$\int_\Omega \left(\rho T^{\text{ref}} \frac{s_{n+1} - s_n}{\Delta t} - \boldsymbol{j}_{n+1}\right) \widehat{T} \, \mathrm{dx} = -\int_{\partial\Omega} q \widehat{T} \, \mathrm{dS} \quad \forall \widehat{T}$$

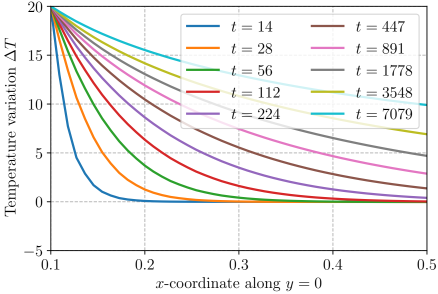
```
sig = problem.get_flux("Stress")
j = problem.get_flux("HeatFlux")
s = problem.get_state_variable("EntropyPerUnitOfMass")
problem.initialize()

s_old = s.copy(deepcopy=True)
v_ = TestFunction(V)
u_, T_ = split(v_)  # Displacement and temperature test functions
eps_ = problem.gradients["Strain"].variation(v_)
mech_residual = dot(sig, eps_)*problem.dx
thermal_residual = (rho*Tref*(s - s_old)/dt*T_ - dot(j, grad(T_)))*problem.dx
problem.residual = mech_residual + thermal_residual
problem.compute_tangent_form()
```

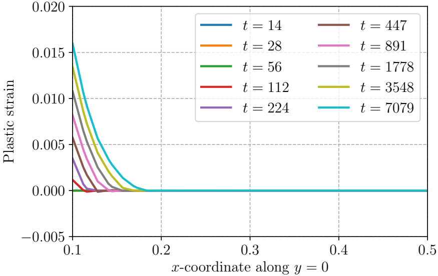
Monolithic thermoelastoplasticity - FEniCS

Variational formulation and implicit backward Euler:

$$\int_\Omega \boldsymbol{\sigma}_{n+1} : \nabla^s \widehat{\boldsymbol{u}} \, \mathrm{dx} = \int_{\partial\Omega} \boldsymbol{T} \cdot \widehat{\boldsymbol{u}} \, \mathrm{dS} \quad \forall \widehat{\boldsymbol{u}}$$

$$\int_\Omega (\rho T^{\mathrm{ref}} \frac{s_{n+1} - s_n}{\Delta t} - \boldsymbol{j}_{n+1}) \widehat{T} \, \mathrm{dx} = - \int_{\partial\Omega} q \widehat{T} \, \mathrm{dS} \quad \forall \widehat{T}$$





To conclude

Conclusions

- working `MGIS/FEniCS` binding although not fully optimized
- behaviour is **separated** from FEniCS implementation (function spaces, strain measures, equilibrium/evolution equations)
- wide range of (complex) applications with **simple** implementation

What's next ?

- tests!
- integration in **dolfin-x**
- multi-materials, plates/shells
- other examples:
  - Cosserat elastoplasticity (Raffaele Russo, Tamara Dancheva)
  - micromorphic crystal plasticity (Julien Sanahuja)
  - periodic polycrystals (orthotropic behaviour support)
  - **any ideas** ?