Numerical tools for automating non-linear computational mechanics simulations

Jeremy Bleyer

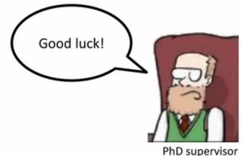
Cours ED SIE Méthodes numériques avancées
March, 18th 2025

Introduction

Yet another finite-element program ?

$\Rightarrow$ need to develop a in-house FE program yourself ?

Introduction



Yet another finite-element program ?

$\Rightarrow$ need to develop a in-house FE program yourself ?

Pros

- free and avoid black-box
- full control for adding new features
- integration with other computational framework
- learning by doing
- educational purposes

Cons

- time-consuming and heavy maintenance (documentation, tests)
- programming language dependence
- not well optimized
- long-term durability
- lack of community

Objectives

Observation: we **are not** computer scientists but we should use developments and tools in this domain to make our life easier

Focus on: improving **efficiency**, **accuracy**, **scalability**, **genericity**, **interoperability**, **sustainability**, **reproducibility** of numerical simulations in mechanics

Objectives

Observation: we **are not** computer scientists but we should use developments and tools in this domain to make our life easier

Focus on: improving **efficiency**, **accuracy**, **scalability**, **genericity**, **interoperability**, **sustainability**, **reproducibility** of numerical simulations in mechanics

New challenges: data assimilation, control/optimization, uncertainty quantification, machine learning integration

Objectives

Observation: we **are not** computer scientists but we should use developments and tools in this domain to make our life easier

Focus on: improving **efficiency**, **accuracy**, **scalability**, **genericity**, **interoperability**, **sustainability**, **reproducibility** of numerical simulations in mechanics

New challenges: data assimilation, control/optimization, uncertainty quantification, machine learning integration

Key notions:

- Code generation
- Just-In-Time (JIT) compilation
- Domain-Specific Language (DSL)
- Automatic Differentiation (AD)
- High-Performance Computing (HPC)
- numerical solvers (linear systems, nonlinear systems, optimization problems)

Outline

1. Automating PDEs with `FEniCSx`
2. Code generation for material constitutive modeling
3. JAX and Automatic Differentiation
4. PDE-based optimisation

Outline

1 **Automating PDEs with `FEniCSx`**
- Introduction
- Linear problems
- Nonlinear problems

2 Code generation for material constitutive modeling

3 JAX and Automatic Differentiation

4 PDE-based optimisation

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, ...

A collection of interoperable components

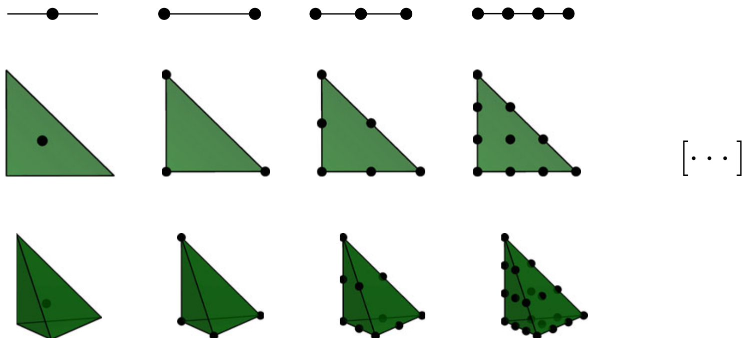
- `basix`: FE definition and tabulation of shape functions, derivatives, mappings
- `ufl` (Unified Form Language), for specifying FE discretizations of PDE in terms of FE variational forms
- `ffcx` (FEniCS Form Compiler), a compiler of UFL forms, generates low-level C code
- `dolfinx`, a C++/Python backend library providing data structures and algorithms for FE meshes, automated FE assembly, and numerical linear algebra

interaction with **PETSc** for linear algebra and linear/nonlinear solvers
fully parallel with MPI

FEniCSx: new implementation of legacy FEniCS

Basix : supported elements

Lagrange elements: arbitrary order, 1D/2D/3D simplices

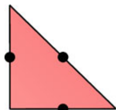
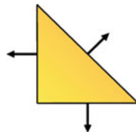
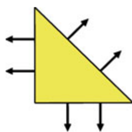
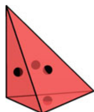
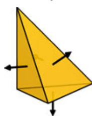
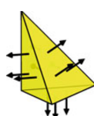


support for quads/hex, higher-order geometry

Soon: composite elements, prisms, mixed meshes

Less standard elements

Crouzeix-Raviart (CR), **Raviart-Thomas** (RT), **Brezzi-Douglas-Marini** (BDM)

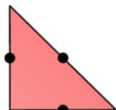
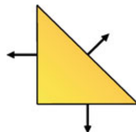
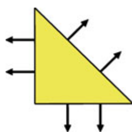
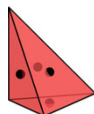
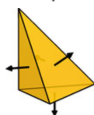
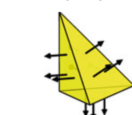







CR_1 RT_1 BDM_1

Usage: mixed methods

Less standard elements

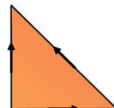
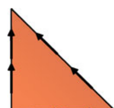
Crouzeix-Raviart (CR), **Raviart-Thomas** (RT), **Brezzi-Douglas-Marini** (BDM)

CR_1 RT_1 BDM_1

Usage: mixed methods

Nédélec

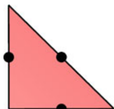
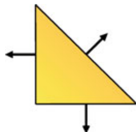
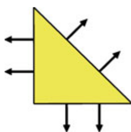
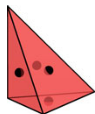
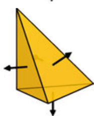



Nédélec (1st/2nd kind)

Usage: electro-magnetism (curl operator)

Less standard elements

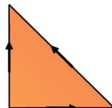
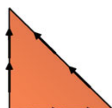
Crouzeix-Raviart (CR), **Raviart-Thomas** (RT), **Brezzi-Douglas-Marini** (BDM)

CR_1 RT_1 BDM_1

Usage: mixed methods

Nédélec

Nédélec (1st/2nd kind)

Usage: electro-magnetism (curl operator)

C^1-elements **not supported** in `dolfinx` (Hermite elements for beams, Morley for plates, strain gradient theory)

A peek under the hood: code generation

Problem: building a mass matrix on a $\mathbb{P}^k$ FE space V
For $(u, v) \in V \times V$

$$m(u, v) = \int_\Omega uv \, d\Omega$$

```
from mpi4py import MPI
import ufl
from dolfinx import fem, mesh

N = 10
domain = mesh.create_unit_square(MPI.COMM_WORLD, N, N)

V = fem.functionspace(domain, ("P", 1, ()))
u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)
m = u * v * ufl.dx

jit_options = {"cache_dir": "./cache"}
m_compiled = fem.form(m, jit_options=jit_options)
```

DEMO

Linear elasticity

Weak form/variational formulation: Find $\boldsymbol{u} \in V$ such that:

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

= weak form of equilibrium and must be supplemented by a constitutive relation
Linear elastic case:

$$\boldsymbol{\sigma}(\boldsymbol{u}) = \mathbb{C} : \nabla^s \boldsymbol{u}$$

Linear elasticity

Weak form/variational formulation: Find $\boldsymbol{u} \in V$ such that:

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

= weak form of equilibrium and must be supplemented by a constitutive relation
Linear elastic case:

$$\boldsymbol{\sigma}(\boldsymbol{u}) = \mathbb{C} : \nabla^s \boldsymbol{u}$$

We end up with a bilinear form k on the lhs and a linear form f on the rhs:
Find $\boldsymbol{u} \in V$ such that:

$$k(\boldsymbol{u}, \boldsymbol{v}) = f(\boldsymbol{v}) \quad \forall \boldsymbol{v} \in V_0$$

FEniCS **automatically** transforms this up to a discrete linear system:

$$[\boldsymbol{K}]\{\boldsymbol{U}\} = \{\boldsymbol{F}\}$$

DEMO

Multi-field variational problems

One of the **big advantages of FEniCS**
e.g. Stokes incompressible fluids
mixed $u - p$ formulation with **Taylor-Hood element** ($\mathbb{P}^2/\mathbb{P}^1$)

```
# Define function space
P2 = VectorElement('P', tetrahedron, 2)
P1 = FiniteElement('P', tetrahedron, 1)
TH = MixedElement(P2, P1)  # or P2*P1
W = FunctionSpace(mesh, TH)

# Define variational problem
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
a = inner(grad(u), grad(v))*dx - p*div(v)*dx + div(u)*q*dx
L = dot(f, v)*dx
```

Multi-field variational problems

One of the **big advantages of FEniCS**
e.g. Stokes incompressible fluids
mixed $u - p$ formulation with **Taylor-Hood element** $(\mathbb{P}^2/\mathbb{P}^1)$

```
# Define function space
P2 = VectorElement('P', tetrahedron, 2)
P1 = FiniteElement('P', tetrahedron, 1)
TH = MixedElement(P2, P1)  # or P2*P1
W = FunctionSpace(mesh, TH)

# Define variational problem
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
a = inner(grad(u), grad(v))*dx - p*div(v)*dx + div(u)*q*dx
L = dot(f, v)*dx
```

MINI element $(\mathbb{P}^1 + \mathcal{B})/\mathbb{P}^1$

```
# Define function space
Pv1 = VectorElement('P', tetrahedron, 1)
B = VectorElement('Bubble', tetrahedron, 3)
W = FunctionSpace(mesh, (Pv1+B)*P1)
```

Reissner-Mindlin plates

Generalized strains

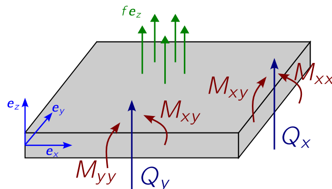
- Bending curvature strain: $\boldsymbol{\chi} = \nabla^s \boldsymbol{\theta}$
- Shear strain: $\boldsymbol{\gamma} = \nabla w - \boldsymbol{\theta}$

Generalized stresses

- Bending moment $\boldsymbol{M}$
- Shear force $\boldsymbol{Q}$

Equilibrium equations

- Vertical equilibrium: $\operatorname{div} \boldsymbol{Q} + f = 0$
- Moment equilibrium: $\operatorname{div} \boldsymbol{M} + \boldsymbol{Q} = 0$



In **weak form**: Find $(w, \boldsymbol{\theta}) \in V$

$$\int_\Omega (\boldsymbol{M} : \nabla^s \widehat{\boldsymbol{\theta}} + \boldsymbol{Q} \cdot (\nabla \widehat{w} - \widehat{\boldsymbol{\theta}})) \mathrm{d}\Omega = \int_\Omega f w \mathrm{d}\Omega \quad \forall \widehat{w}, \widehat{\boldsymbol{\theta}} \in V$$

DEMO

Time-dependent problems

No built-in support for time discretization schemes ⇒ should be handled in the variational formulation e.g. transient heat equation:

$$\rho c \frac{\partial T}{\partial t} + \mathrm{div}(-k\nabla T) = 0$$

Weak form:

$$\int_\Omega \left(\rho c \frac{\partial T}{\partial t}\widehat{T} + k\nabla T \cdot \nabla \widehat{T}\right) \,\mathrm{d}\Omega = 0$$

Time-dependent problems

No built-in support for time discretization schemes $\Rightarrow$ should be handled in the variational formulation e.g. transient heat equation:

$$\rho c \frac{\partial T}{\partial t} + \operatorname{div}(-k\nabla T) = 0$$

Weak form:

$$\int_\Omega \left(\rho c \frac{\partial T}{\partial t}\widehat{T} + k\nabla T \cdot \nabla \widehat{T}\right) \,\mathrm{d}\Omega = 0$$

Implicit Euler: $\dfrac{\partial T}{\partial t} \approx \dfrac{T - T_n}{\Delta t}$

$$\int_\Omega \left(\rho c \frac{T - T_n}{\Delta t}\widehat{T} + k\nabla T \cdot \nabla \widehat{T}\right) \,\mathrm{d}\Omega = 0$$

$$\int_\Omega \left(\rho c T \widehat{T} + \Delta t k\nabla T \cdot \nabla \widehat{T}\right) \,\mathrm{d}\Omega = \int_\Omega \rho c T_n \widehat{T} \,\mathrm{d}\Omega$$

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:

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

$$\text{residual} \quad R(\boldsymbol{u}) = \int_\Omega \frac{\partial \psi}{\partial \boldsymbol{F}} : \nabla \boldsymbol{v} \,\mathrm{d}\Omega - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \,\mathrm{d}\Omega - \int_{\partial\Omega_N} \boldsymbol{T} \cdot \boldsymbol{v} \,\mathrm{dS} = 0 \quad \forall \boldsymbol{v} \in V_0$$

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

solvers: built-in Newton or PETSc SNES

Hyperelasticity

Simple definition using UFL **automatic differentiation**

DEMO

Hyperelasticity

Simple definition using UFL **automatic differentiation**

DEMO

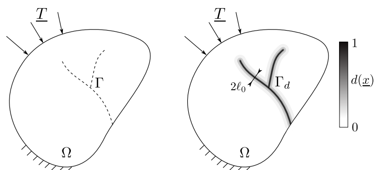
Hot-to-cold analysis *(turbine blade shape matching)*

Find reference configuration Ω_0 from **known** equilibrium configuration Ω_t

$$\text{Find } \boldsymbol{X} \in \Omega_0 \text{ s.t. } \boldsymbol{x} = \boldsymbol{X} + \boldsymbol{u}(\boldsymbol{x}) \quad \boldsymbol{x} \in \Omega_t$$

$$\boldsymbol{F} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} = \left(\frac{\partial \boldsymbol{X}}{\partial \boldsymbol{x}}\right)^{-1} = (\boldsymbol{I} - \nabla_x \boldsymbol{u})^{-1}$$

Phase-field approach to fracture



Two-fields minimum principle [Bourdin et al., 2000]: $u(t)$, $d(t)$ minimizes the total energy:

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

with the irreversibility condition $\dot{d} \geq 0$
where:

$$\boxed{\mathcal{E}_{pot}(\boldsymbol{u}, d) = \int_\Omega (1-d)^2 \frac{1}{2} \boldsymbol{\varepsilon} : \mathbb{C} : \boldsymbol{\varepsilon} \, \mathrm{d}\Omega - W_{ext}(\boldsymbol{u})}$$

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

Numerical implementation

Classical strategy: **alternate minimization**

Numerical implementation

Classical strategy: **alternate minimization**
at time t_{n+1}, we know the past solution (u_n, d_n), we iterate:

$$
\begin{aligned}
& u_{n+1}^0 = u_n \text{ and } d_{n+1}^0 = d_n \\
& \text{for } i = 1, \dots, N_{\text{iter max}}: \\
& \quad u_{n+1}^i = \arg\min_{v} \mathcal{E}_{tot}(v, d_{n+1}^i) \qquad (1) \\
& \quad d_{n+1}^i = \arg\min_{d_n \leq d \leq 1} \mathcal{E}_{tot}(u_{n+1}^i, d) \qquad (2) \\
& \quad \text{stop if } \|(u_{n+1}^i, d_{n+1}^i) - (u_{n+1}^{i-1}, d_{n+1}^{i-1})\| \leq \texttt{tol}
\end{aligned}
$$

Numerical implementation

Classical strategy: **alternate minimization**
at time t_{n+1}, we know the past solution (u_n, d_n), we iterate:

$$
\begin{aligned}
& u_{n+1}^0 = u_n \text{ and } d_{n+1}^0 = d_n \\
& \text{for } i = 1, \ldots, N_{\text{iter max}} : \\
& \quad u_{n+1}^i = \arg\min_v \mathcal{E}_{tot}(v, d_{n+1}^i) \qquad (1) \\
& \quad d_{n+1}^i = \arg\min_{d_n \leq d \leq 1} \mathcal{E}_{tot}(u_{n+1}^i, d) \qquad (2) \\
& \quad \text{stop if } \|(u_{n+1}^i, d_{n+1}^i) - (u_{n+1}^{i-1}, d_{n+1}^{i-1})\| \leq \texttt{tol}
\end{aligned}
$$

Problem (1) = **standard elasticity problem** with fixed value of $d = d_{n+1}^i$

Numerical implementation

Classical strategy: **alternate minimization**
at time t_{n+1}, we know the past solution (u_n, d_n), we iterate:

$$
\begin{aligned}
& u^0_{n+1} = u_n \text{ and } d^0_{n+1} = d_n \\
& \text{for } i = 1, \ldots, N_{\text{iter max}} : \\
& \quad u^i_{n+1} = \arg\min_v \mathcal{E}_{tot}(v, d^i_{n+1}) \qquad (1) \\
& \quad d^i_{n+1} = \arg\min_{d_n \leq d \leq 1} \mathcal{E}_{tot}(u^i_{n+1}, d) \qquad (2) \\
& \quad \text{stop if } \|(u^i_{n+1}, d^i_{n+1}) - (u^{i-1}_{n+1}, d^{i-1}_{n+1})\| \leq \texttt{tol}
\end{aligned}
$$

Problem (1) = **standard elasticity problem** with fixed value of $d = d^i_{n+1}$

Problem (2) for AT1/AT2 = **minimizing a quadratic energy** in terms of d with bound constraints $d_n \leq d \leq 1$
$\Rightarrow$ there exist **dedicated solvers** (e.g. TAO distributed with PETSc)

Implementation

```
elastic_energy = (1-d)**2 * psi(u) * dx
fracture_energy = Gc / cw/l0 * (d**2 + l0**2 * dot(grad(d), grad(d))) * dx
total_energy = elastic_energy + fracture_energy

F_u = derivative(elastic_energy, u, v)
J_u = derivative(F_u, u, u_)

problem_u = NonlinearProblem(F_u, u, bcs)
u_solver = NewtonSolver(domain.comm, problem_u)

# first derivative of energy with respect to d
F_dam = derivative(total_energy, d, d_)
# second derivative of energy with respect to d
J_dam = derivative(F_dam, d, dd)
# Definition of the optimisation problem with respect to d
damage_problem = TAOProblem(total_energy, F_dam, J_dam, d, bc_d)

solver_d_tao = PETSc.TAO().create()
solver_d_tao.setType("tron")
solver_d_tao.setObjective(damage_problem.f)
solver_d_tao.setGradient(damage_problem.F, b)
solver_d_tao.setHessian(damage_problem.J, J)

# We set the bounds
solver_d_tao.setVariableBounds(dlb.vector, dub.vector)
```

Implementation

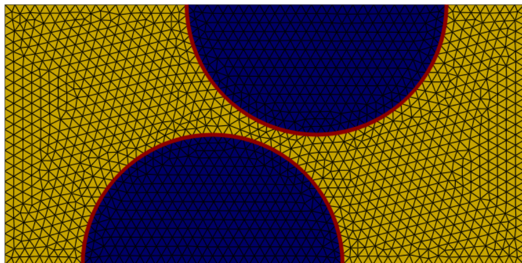
```
for i, t in enumerate(load_steps[1:]):
    # update bcs

    niter = 0
    for niter in range(Nitermax):
        # Solve displacement
        u_solver.solve(u)
        # Compute new damage
        solver_d_tao.solve(d.vector)

        # check error and update
        L2_error = fem.form(inner(d - dold, d - dold) * dx)
        error_L2 = np.sqrt(fem.assemble_scalar(L2_error)) / vol

        # Update damage
        d.vector.copy(dold.vector)

        if error_L2 < tol:
            break
    else:
        warnings.warn("Too many iterations in fixed point algorithm")

    # Update lower bound to account for irreversibility
    d.vector.copy(dlb.vector)
```

New feature: mixed domains

`dolfinx.mesh.create_submesh`
PDEs can now be solved on **subdomains** of **same dimension** or of **codimension 1** (1D/2D or 2D/3D)

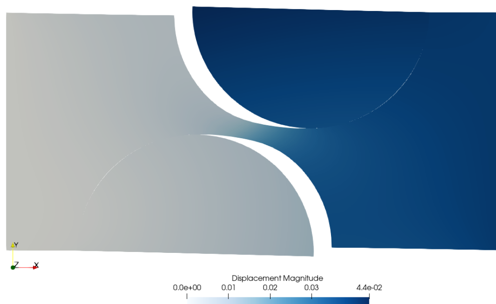
Example: CZM on an interface

$$\int_\Omega \boldsymbol{\sigma}(\nabla^s \boldsymbol{u}) : \nabla^s \boldsymbol{v} \,\mathrm{d}\Omega + \int_\Gamma \boldsymbol{T}([\![\boldsymbol{u}]\!], d) \cdot [\![\boldsymbol{v}]\!] \,\mathrm{dS} = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \,\mathrm{d}\Omega \quad \forall \boldsymbol{v} \in V_0$$



2 subdomains for matrix and inclusion (displacements) + 1 interface subdomain for damage variables

New feature: mixed domains



Other hyperelastic models: UFL representable or not ?

$I_1 = \mathrm{tr}(\boldsymbol{C})$, $I_2 = \dfrac{1}{2}(I_1^2 - \mathrm{tr}(\boldsymbol{C}^2))$, $J = \det \boldsymbol{F}$, $\boldsymbol{E} = \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I})$

- Saint-Venant Kirchhoff:

$$\psi(\boldsymbol{C}) = \frac{\lambda}{2}\mathrm{tr}(\boldsymbol{E})^2 + \mu \boldsymbol{E} : \boldsymbol{E} \quad ✓$$

Other hyperelastic models: UFL representable or not ?

$I_1 = \operatorname{tr}(\boldsymbol{C}),\ I_2 = \dfrac{1}{2}(I_1^2 - \operatorname{tr}(\boldsymbol{C}^2)),\ J = \det \boldsymbol{F},\ \boldsymbol{E} = \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I})$

- Saint-Venant Kirchhoff:

$$\psi(\boldsymbol{C}) = \frac{\lambda}{2}\operatorname{tr}(\boldsymbol{E})^2 + \mu \boldsymbol{E} : \boldsymbol{E} \quad ✓$$

- Arruda-Boyce:

$$\psi(\boldsymbol{C}) = C_1 \sum_{n=1}^{N} \alpha_n (I_1^n - 3^n) \quad ✓$$

Other hyperelastic models: UFL representable or not ?

$I_1 = \operatorname{tr}(\boldsymbol{C})$, $I_2 = \dfrac{1}{2}(I_1^2 - \operatorname{tr}(\boldsymbol{C}^2))$, $J = \det \boldsymbol{F}$, $\boldsymbol{E} = \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I})$

- Saint-Venant Kirchhoff:

$$\psi(\boldsymbol{C}) = \frac{\lambda}{2}\operatorname{tr}(\boldsymbol{E})^2 + \mu \boldsymbol{E} : \boldsymbol{E} \quad \checkmark$$

- Arruda-Boyce:

$$\psi(\boldsymbol{C}) = C_1 \sum_{n=1}^{N} \alpha_n (I_1^n - 3^n) \quad \checkmark$$

- Mooney-Rivlin:

$$\psi(\boldsymbol{C}) = C_1(J^{-2/3} I_1 - 3) + C_2(J^{-4/3} I_2 - 3) \quad \checkmark$$

Other hyperelastic models: UFL representable or not ?

$I_1 = \mathrm{tr}(\boldsymbol{C}),\ I_2 = \dfrac{1}{2}(I_1^2 - \mathrm{tr}(\boldsymbol{C}^2)),\ J = \det \boldsymbol{F},\ \boldsymbol{E} = \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I})$

- Saint-Venant Kirchhoff:

$$\psi(\boldsymbol{C}) = \frac{\lambda}{2}\mathrm{tr}(\boldsymbol{E})^2 + \mu \boldsymbol{E} : \boldsymbol{E} \quad ✓$$

- Arruda-Boyce:

$$\psi(\boldsymbol{C}) = C_1 \sum_{n=1}^{N} \alpha_n (I_1^n - 3^n) \quad ✓$$

- Mooney-Rivlin:

$$\psi(\boldsymbol{C}) = C_1(J^{-2/3}I_1 - 3) + C_2(J^{-4/3}I_2 - 3) \quad ✓$$

- Ogden:

$$\psi(\boldsymbol{C}) = \sum_{n=1}^{N} \frac{\mu_n}{\alpha_n}(\lambda_1^{\alpha_n} + \lambda_2^{\alpha_n} + \lambda_3^{\alpha_n}) \quad ✗$$

need closed-form expression for λ_i in 3D, $\partial\psi/\partial\boldsymbol{C}, \partial^2\psi/\partial\boldsymbol{C}\partial\boldsymbol{C}$???

von Mises plasticity

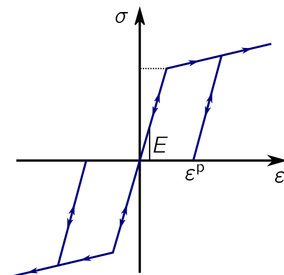
Isotropic hardening $h(p)$:

$$\boldsymbol{\sigma} = \mathbb{C} : (\varepsilon - \varepsilon^p)$$

$$\dot{\varepsilon}^p = \dot{p}\frac{\boldsymbol{s}}{\sigma_{eq}}$$

$$f(\boldsymbol{\sigma}) = \sqrt{\frac{3}{2}\boldsymbol{s} : \boldsymbol{s}} - (\sigma_0 + h(p)) \leq 0$$

$$\dot{p} \geq 0, \quad \dot{p}f(\boldsymbol{\sigma}) = 0$$



von Mises plasticity

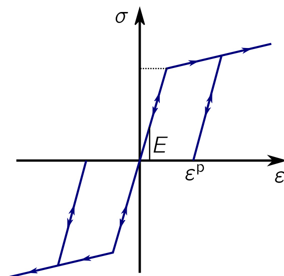
Isotropic hardening $h(p)$:

$$\boldsymbol{\sigma} = \mathbb{C} : (\varepsilon - \varepsilon^p)$$

$$\dot{\varepsilon}^p = \dot{p}\frac{\boldsymbol{s}}{\sigma_{eq}}$$

$$f(\boldsymbol{\sigma}) = \sqrt{\frac{3}{2}\boldsymbol{s}:\boldsymbol{s}} - (\sigma_0 + h(p)) \leq 0$$

$$\dot{p} \geq 0, \quad \dot{p}f(\boldsymbol{\sigma}) = 0$$



Strain-driven strategy: $\Delta\varepsilon, p_n, \boldsymbol{\sigma}_n \longrightarrow$ Constitutive relation $\longrightarrow p_{n+1}, \boldsymbol{\sigma}_{n+1}$

von Mises plasticity

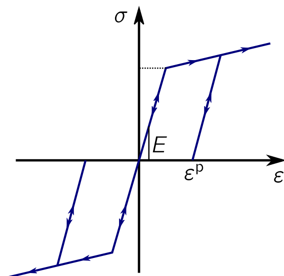
Isotropic hardening $h(p)$:

$$\boldsymbol{\sigma} = \mathbb{C} : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^p)$$

$$\dot{\boldsymbol{\varepsilon}}^p = \dot{p}\frac{\boldsymbol{s}}{\sigma_{eq}}$$

$$f(\boldsymbol{\sigma}) = \sqrt{\frac{3}{2}\boldsymbol{s}:\boldsymbol{s}} - (\sigma_0 + h(p)) \leq 0$$

$$\dot{p} \geq 0, \quad \dot{p}f(\boldsymbol{\sigma}) = 0$$



Strain-driven strategy: $\Delta\boldsymbol{\varepsilon}, p_n, \boldsymbol{\sigma}_n \longrightarrow$ Constitutive relation $\longrightarrow p_{n+1}, \boldsymbol{\sigma}_{n+1}$

Newton method:

$$\int_\Omega \boldsymbol{\sigma}(\boldsymbol{\varepsilon}_n + \Delta\boldsymbol{\varepsilon}) : \nabla^s \boldsymbol{v} \,\mathrm{d}\Omega - L(\boldsymbol{v}) = 0 \quad \forall \boldsymbol{v} \in V$$

we need the tangent matrix $\mathbb{C}_t = \partial\boldsymbol{\sigma}/\partial\Delta\boldsymbol{\varepsilon}$

von Mises plasticity

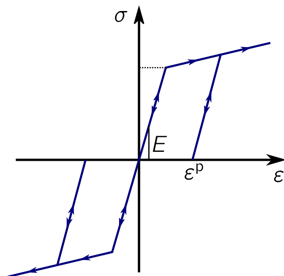
Isotropic hardening $h(p)$:

$$\boldsymbol{\sigma} = \mathbb{C} : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^p)$$

$$\dot{\boldsymbol{\varepsilon}}^p = \dot{p}\frac{\boldsymbol{s}}{\sigma_{eq}}$$

$$f(\boldsymbol{\sigma}) = \sqrt{\frac{3}{2}\boldsymbol{s}:\boldsymbol{s}} - (\sigma_0 + h(p)) \leq 0$$

$$\dot{p} \geq 0, \quad \dot{p}f(\boldsymbol{\sigma}) = 0$$



Strain-driven strategy: $\Delta\boldsymbol{\varepsilon}, p_n, \boldsymbol{\sigma}_n \longrightarrow$ Constitutive relation $\longrightarrow p_{n+1}, \boldsymbol{\sigma}_{n+1}$

Newton method:

$$\int_\Omega \boldsymbol{\sigma}(\boldsymbol{\varepsilon}_n + \Delta\boldsymbol{\varepsilon}) : \nabla^s \boldsymbol{v} \, \mathrm{d}\Omega - L(\boldsymbol{v}) = 0 \quad \forall \boldsymbol{v} \in V$$

we need the tangent matrix $\mathbb{C}_t = \partial\boldsymbol{\sigma}/\partial\Delta\boldsymbol{\varepsilon}$

$\boldsymbol{\sigma}$ and $\mathbb{C}_t$ **not UFL-representable**

Outline

1 Automating PDEs with FEniCSx

2 Code generation for material constitutive modeling
- Introduction
- The MFront code generator
- On the FEniCSx side

3 JAX and Automatic Differentiation

4 PDE-based optimisation

Loi de comportement

Code_Aster (Fortran)

```
subroutine nmccam(fami, kpg, ksp, ndim, &
                  typmod, imate, carcri, &
                  deps, sigm, pcrm, option, sigp, &
                  pcrp, dsidep, retcom)
!
    implicit none

!     -- 3 CALCUL DE DEPSMO ET DEPSDV :
!     --------------------------------
    if (cplan) then
        call utmess('F', 'ALGORITH6_63')
    end if
    depsmo = 0.d0
    do k = 1, ndimsi
        depsth(k) = deps(k)
    end do
    do k = 1, 3
        depsth(k) = depsth(k)-coef
        depsmo = depsmo+depsth(k)
    end do
    depsmo = -depsmo
    do k = 1, ndimsi
        depsdv(k) = depsth(k)+depsmo/3.d0*kron(k)
    end do
!
!     -- 4 CALCUL DE SIGMMO, SIGMDV, SIGEL,SIMOEL,SIELEQ, SIEQM :
```

Loi de comportement

Cast3m (ESOPE):

```
C CAMCLA    SOURCE    FANDEUR   22/05/02    21:15:02     11359
      SUBROUTINE CAMCLA(SIG0,NSTRS,DEPST,VAR0,NVARI,XMAT,NCOMAT,XCAR,
     .                  SIGF,VARF,DEFP,PRECIS,MFR,KERRE)

*
*   ON CALCULE LES CONTRAINTES FINALES EN ELASTIQUE
*
      XK0=(1.D0+E0)/XKAPA
      TRACEP= TRACE(DEPST)
      TRASI0 = (TRACE(SIG0))/3.D0  - COHE
      FAC = TRASI0 * (EXP(-XK0*TRACEP) -1.D0)
      STOT(1)= SIG0(1)+G2*(DEPST(1)-TRACEP/3.D0) +FAC
      STOT(2)= SIG0(2)+G2*(DEPST(2)-TRACEP/3.D0) +FAC
      STOT(3)= SIG0(3)+G2*(DEPST(3)-TRACEP/3.D0) +FAC
      STOT(4)= SIG0(4)+G*DEPST(4)
      IF(IFOUR.GE.1) THEN
        STOT(5)= SIG0(5)+G*DEPST(5)
        STOT(6)= SIG0(6)+G*DEPST(6)
      ELSE
        STOT(5)= 0.D0
        STOT(6)= 0.D0
      ENDIF
*
      IF(IIMPI.EQ.33) THEN
        WRITE(IOIMP,77875) XK0,TRACEP
77875   FORMAT(1X,'  XK0 = ',1PE12.5,2X,'TRACEP = ',1PE12.5/)
```

Loi de comportement

https://github.com/deryckchan/cam_clay_element (Matlab)

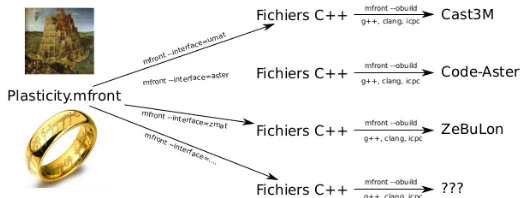
```
%% Strain stepping: set de before running this script; it gives ds
% Impose_Strain_MCC

% Calculate F. If F >= 0 already then update pc and do plastic flow
F = q*q/M/M - pc*p + p*p;
vCSL = vatm - (lambda - kappa) * log(2) - lambda * log(p / patm);
vCSLmarginp = 0.002; % Numerical hack to prevent too much flip-flopping between wet and dry
     side for no good reason
vCSLmarginm = -0.001;
% Derived state variables: Plastic

if F > 0
    % Split dry side and wet side!
    if (v > vCSL + vCSLmarginp) % Dry side. Bubble is expanding, just expand bubble
        % update pc based on new values of p and q at the start of this stage
        pc = q*q/M/M/p + p;
    elseif (v < vCSL - vCSLmarginm)
        % Wet side. Bubble is contracting, use volume
        pc = patm * exp((vatm - v - kappa * log(p / patm)) / (lambda - kappa));
    end % Else: very close to critical state already, stop updating pc

    % Calculate plastic terms
    dFds = (6/M/M).*s.*[0 0 0 1 1 1]' + ((2*p-pc)/3 + (3/M/M).*(s-p)).*[1 1 1 0 0 0]';
    dFdWdWde = (-p * pc * vstart / (lambda - kappa)) .* [1 1 1 0 0 0]; % Using vstart gives
     more stable results than v(instantaneous)
```

Loi de comportement

- decouple **FE solvers** from **material library**
- **inter-operability**: easy integration of a material library in various solvers (industrial and academic)
- improve quality assurance, long-term sustainability, collaboration

Introduction

https://thelfer.github.io/tfel/web/index.html



- `MFront` = a **code generation tool** dedicated to material knowledge (material properties, mechanical behaviours, point-wise models) :
  Support for small and finite strain behaviours, cohesive zone models, generalised behaviours (non local and or multiphysics).
- Main goals :
  - Numerical efficiency
  - Inter-operability: **specific** (`Cast3M`, `code_aster`, `Europlexus`, `Abaqus`, `Zebulon`) and **generic interfaces**
  - Ease of use (**Domain-Specific Languages**)
- developed at CEA: main developer = Thomas Helfer

Ecosystem

- **TFEL/MFront**:
  - TFEL C++ libraries: Math/Material
  - **Behaviors** Domain-Specific Languages: `Default`, `Implicit`, `DefaultFiniteStrain`, `DefaultCZM`, `DefaultGenericBehavior`, etc.
  - **Bricks**: `StandardElasticity`, `StandardElastoViscoplasticity`, `FiniteStrainSingleCrystal`

```
@DSL Implicit;
@Behaviour PerfectPlasticity;

@Epsilon 1.e-14;
@Theta 1;

@Brick StandardElastoViscoPlasticity{
  stress_potential : "Hooke" {young_modulus : 200e9, poisson_ratio : 0.3},
  inelastic_flow : "Plastic" {
    criterion : "Mises",
    isotropic_hardening : "Linear" {R0 : 150e6},
  }
};
```

- **MTest**: testing tool
- **Gallery**: documented examples
- **MFrontGenericInterfaceSupport**: open-source library to interact with a compiled behavior in Python, Fortran, C, Julia, etc. [Helfer, Thomas, et al. "The MFrontGenericInterfaceSupport project." Journal of Open Source Software 5.48 (2020): 1-8.]

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
```

$$\boldsymbol{\sigma} = \mathbb{C} : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^{vp})$$
$$\dot{\boldsymbol{\varepsilon}}^{vp} = \dot{p}\boldsymbol{n}$$
$$\boldsymbol{n} = \frac{\boldsymbol{s}}{\sigma_{eq}}$$
$$\dot{p} = A\sigma_{eq}^m$$

Unicode support

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 at $t_n + \theta\Delta t$:

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

Unicode support

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
```

Unicode support

Implicit system at $t_n + \theta\Delta t$:

$$f_{\varepsilon^e} = \Delta\varepsilon^e - \Delta\varepsilon + \Delta p\boldsymbol{n} = 0$$

$$f_p = \Delta p - A(\sigma_{eq})^n = 0$$

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

Advanced aspects

- **Finite strain**: solvers require different "forms" of the tangent operator e.g. $\dfrac{d\boldsymbol{\sigma}}{d\boldsymbol{d}}, \dfrac{d\boldsymbol{P}}{d\boldsymbol{F}}, \dfrac{d\boldsymbol{S}}{d\boldsymbol{E}}$ and many others... $\Rightarrow$ MFront provides all conversion methods internally
- support for different modelling hypotheses: `PlaneStrain, Tridimensional, Axisymmetric, etc.` (optimized code/hypothesis)
- numerical jacobian for implicit systems
- compile-time dimensional analysis
- recently, support for **generalized behaviours**

Mechanics:

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

Advanced aspects

- **Finite strain**: solvers require different "forms" of the tangent operator e.g. $\dfrac{d\boldsymbol{\sigma}}{d\boldsymbol{d}}, \dfrac{d\boldsymbol{P}}{d\boldsymbol{F}}, \dfrac{d\boldsymbol{S}}{d\boldsymbol{E}}$ and many others... $\Rightarrow$ MFront provides all conversion methods internally
- support for different modelling hypotheses: `PlaneStrain`, `Tridimensional`, `Axisymmetric`, `etc.` (optimized code/hypothesis)
- numerical jacobian for implicit systems
- compile-time dimensional analysis
- recently, support for **generalized 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:

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

$\boldsymbol{g}^j$ 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.)

`dolfinx_materials`: Python package for material behaviors

https://github.com/bleyerj/dolfinx_materials

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

`dolfinx_materials`: Python package for material behaviors

https://github.com/bleyerj/dolfinx_materials

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

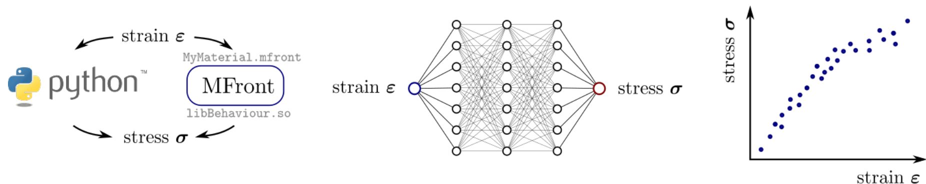
https://github.com/bleyerj/dolfinx_materials

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

```
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

```
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 $\dfrac{\mathrm{d}\texttt{"Stress"}}{\mathrm{d}\texttt{"Strain"}}$.

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

FEniCSxMFront integration

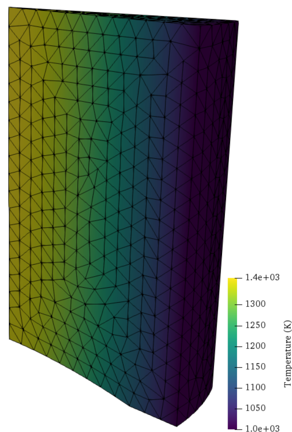
MFrontMaterial class for loading a MFront library, calling the behaviour integration and giving access to fluxes, state variables and tangent operators

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**):

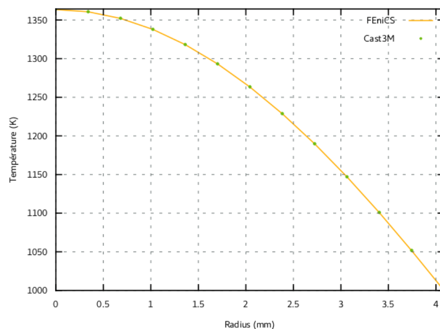
```
mat_prop = {"YoungModulus": E, "PoissonRatio": nu,
            "HardeningSlope": H, "YieldStrength": sig0}
material = MFrontMaterial("src/libBehaviour.so",
               "IsotropicLinearHardeningPlasticity",
               hypothesis="plane_strain",
               material_properties=mat_prop)

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



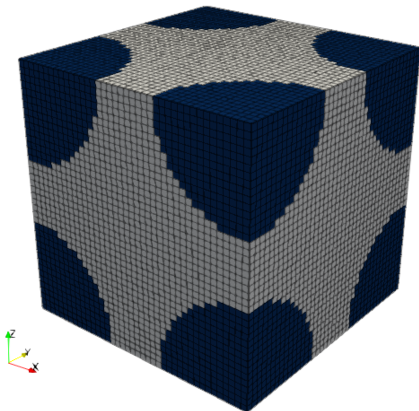
quad_deg	dolfinx/MFront	dolfinx
2	15.76 s	15.22 s
5	16.53 s	15.56 s



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 $\overline{C} = J^{-2/3}\boldsymbol{F}^\mathsf{T}\boldsymbol{F}$



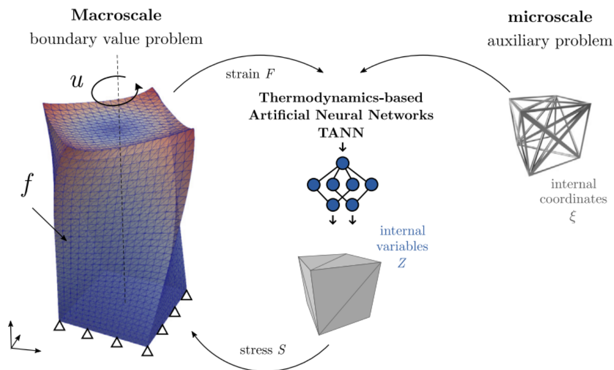
Ogden hyperelasticity

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

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**

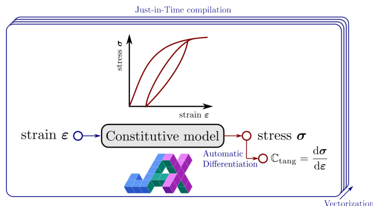
FEM x TANN in `dolfinx_materials`

Underlying microstructure = 3D truss

Outline

1. Automating PDEs with FEniCSx
2. Code generation for material constitutive modeling
3. **JAX and Automatic Differentiation**
   - JAX for constitutive modeling
   - Implicit Automatic Differentiation
   - Material model calibration
4. PDE-based optimisation

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)
```

A simple example

Linear viscoelasticity DEMO

```
class LinearViscoElasticity(JAXMaterial):
    [...]

    @property
    def internal_state_variables(self):
        return {"epsv": 6}

    @tangent_AD
    def constitutive_update(self, eps, state, dt):
        epsv_old = state["epsv"]
        eps_old = state["Strain"]
        deps = eps - eps_old

        epsv_new = (
            eps
            + jnp.exp(-dt / self.tau) * (epsv_old - eps_old)
            - jnp.exp(-dt / 2 / self.tau) * deps
        )

        sig = self.branch0.C @ eps + self.branch1.C @ (eps - epsv_new)
        state["epsv"] = epsv_new
        state["Strain"] = eps
        state["Stress"] = sig
        return sig, state
```

What is Automatic Differentiation ?

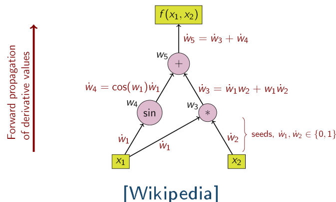
- **Numerical differentiation**: $f'(x) \approx \dfrac{f(x+h) - f(x)}{h}$ with e.g. $h = 10^{-6}$
  truncation/rounding errors, $O(dim)$ **evauations**

What is Automatic Differentiation ?

- **Numerical differentiation**: $f'(x) \approx \dfrac{f(x+h) - f(x)}{h}$ with e.g. $h = 10^{-6}$
  truncation/rounding errors, $O(dim)$ **evauations**
- **Symbolic differentatiation**: f represented as an **expression graph**, generates **another expression graph of the derivative**
  expression swell, duplicate operations, no-closed form expression

What is Automatic Differentiation ?

- **Numerical differentiation**: $f'(x) \approx \dfrac{f(x+h)-f(x)}{h}$ with e.g. $h = 10^{-6}$
  truncation/rounding errors, $O(dim)$ **evauations**
- **Symbolic differentatiation**: f represented as an **expression graph**, generates **another expression graph of the derivative**
  expression swell, duplicate operations, no-closed form expression
- **Automatic differentiation**: operates **directly on the computer program**, no symbolic representation (numerical evaluation only), **exact**
  forward and **reverse** mode (**back-propagation** in ML)



[Wikipedia]

```
def f(x):
    [...]
    for i in range(n):
        [...]
```

```
def f(x):
    [...]
    if cond:
        [...]
    else:
        [...]
```

```
def f(x):
    [...]
    while cond:
        [...]
```

Differentiating through elastoplasticity

von Mises plasticity with nonlinear isotropic hardening $R(p)$

Return mapping algorithm

Elastic predictor $\boldsymbol{\sigma}_\text{elas} = \boldsymbol{\sigma}_n + \mathbb{C} : \Delta\varepsilon$
$f_\text{elas} = \sigma_\text{eq}^\text{elas} - R(p_n)$

- **if** $f_\text{elas} < 0$: $\boldsymbol{\sigma} = \boldsymbol{\sigma}_\text{elas}$ and $\Delta p = 0$
- **else** $\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_\text{elas} - 2\mu\Delta\varepsilon^\text{p}$ with $\Delta\varepsilon^\text{p} = \Delta p \dfrac{3}{2\sigma_\text{eq}^\text{elas}}\boldsymbol{s}_\text{elas}$

$$\textbf{Solve } r(\Delta p) = \sigma_\text{eq}^\text{elas} - 3\mu\Delta p - R(p_n + \Delta p) = 0 \quad (1)$$

e.g. using fixed-point algorithm, Newton method, bisection, etc.

Every step is differentiable with AD, **except** (1).

Differentiating through elastoplasticity

von Mises plasticity with nonlinear isotropic hardening $R(p)$

Return mapping algorithm

Elastic predictor $\boldsymbol{\sigma}_\text{elas} = \boldsymbol{\sigma}_n + \mathbb{C} : \Delta\boldsymbol{\varepsilon}$
$f_\text{elas} = \sigma_\text{eq}^\text{elas} - R(p_n)$

- **if** $f_\text{elas} < 0$: $\boldsymbol{\sigma} = \boldsymbol{\sigma}_\text{elas}$ and $\Delta p = 0$
- **else** $\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_\text{elas} - 2\mu\Delta\boldsymbol{\varepsilon}^\text{p}$ with $\Delta\boldsymbol{\varepsilon}^\text{p} = \Delta p \dfrac{3}{2\sigma_\text{eq}^\text{elas}}\boldsymbol{s}_\text{elas}$

$$\textbf{Solve } r(\Delta p) = \sigma_\text{eq}^\text{elas} - 3\mu\Delta p - R(p_n + \Delta p) = 0 \tag{1}$$

e.g. using fixed-point algorithm, Newton method, bisection, etc.

Every step is differentiable with AD, **except** (1).

Algorithm unrolling

Any algorithm used to solve (1) can be written in JAX using loops, conditionals, etc.
We can **differentiate through the algorithm** (*unrolling the algorithm iterations*).

Implicit automatic differentiation [Blondel et al., 2022]

We can leverage instead the **implicit function theorem**
e.g. **root finding**: Find x_θ s.t. $F(x_\theta;\theta) = 0$

Implicit automatic differentiation [Blondel et al., 2022]

We can leverage instead the **implicit function theorem**
e.g. **root finding**: Find x_θ s.t. $F(x_\theta; \theta) = 0$
To find $\partial_\theta x_\theta$, we differentiate the equation so that:

$$\begin{aligned} &\partial_x F \partial_\theta x_\theta + \partial_\theta F = 0 \\ &\quad \Rightarrow \quad \partial_\theta x_\theta = -[\partial_x F]^{-1} \partial_\theta F \end{aligned}$$

need only to solve a **linear system** for the **jacobian matrix** $[\partial_x F]$
the derivative computation becomes **independent from the algorithm** used to solve the nonlinear system, can use AD to form the jacobian $[\partial_x F]$

Implicit automatic differentiation [Blondel et al., 2022]

We can leverage instead the **implicit function theorem**
e.g. **root finding**: Find x_θ s.t. $F(x_\theta;\theta)=0$
To find $\partial_\theta x_\theta$, we differentiate the equation so that:

$$\begin{aligned} \partial_x F \partial_\theta x_\theta + \partial_\theta F &= 0 \\ \Rightarrow \quad \partial_\theta x_\theta &= -[\partial_x F]^{-1}\partial_\theta F \end{aligned}$$

need only to solve a **linear system** for the **jacobian matrix** $[\partial_x F]$
the derivative computation becomes **independent from the algorithm** used to solve the nonlinear system, can use AD to form the jacobian $[\partial_x F]$

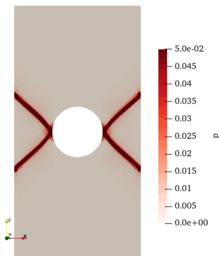
Implementation of `JAXNewton`

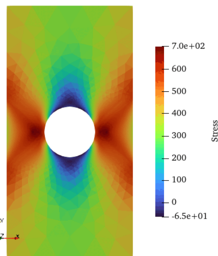
```
class JAXNewton:
    """A tiny Newton solver implemented in JAX.
       Derivatives are computed via custom implicit differentiation."""

    def solve(self, x):
        solve = lambda f, x: newton_solve(x, f, jax.jacfwd(f), self.params)

        tangent_solve = lambda g, y: _solve_linear_system(x, jax.jacfwd(g)(y), y)

        return jax.lax.custom_root(self.r, x, solve, tangent_solve, has_aux=True)
```

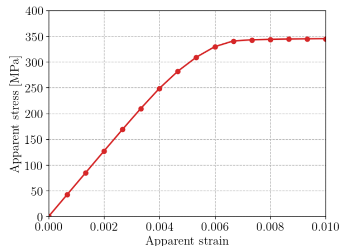
Small-strain elastoplasticity

```python
@tangent_AD
def constitutive_update(self, eps, state, dt):
    deps = eps - state["Strain"]
    p_old = state["p"]

    mu = self.elastic_model.mu
    sig_el = state["Stress"] + self.elastic_model.C @ deps
    sig_eq_el = jnp.clip(self.equivalent_stress(sig_el), a_min=1e-8)
    n_el = dev(sig_el) / sig_eq_el
    yield_criterion = sig_eq_el - self.yield_stress(p_old)

    deps_p_elastic = lambda dp: jnp.zeros(6)
    deps_p_plastic = lambda dp: 3 / 2 * n_el * dp
    def deps_p(dp, yield_criterion):
        return jax.lax.cond(yield_criterion < 0.0, deps_p_elastic, deps_p_plastic, dp)

    def r(dp):
        r_elastic = lambda dp: dp
        r_plastic = lambda dp: sig_eq_el - 3 * mu * dp - self.yield_stress(p_old + dp)
        return jax.lax.cond(yield_criterion < 0.0, r_elastic, r_plastic, dp)

    solver = JAXNewton(r)
    dp, data = solver.solve(0.0)

    sig = sig_el - 2 * mu * deps_p(dp, yield_criterion)
    state["p"] += dp
    return sig, state
```

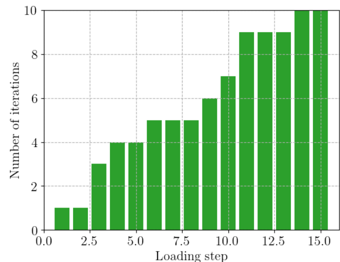
Small-strain elastoplasticity





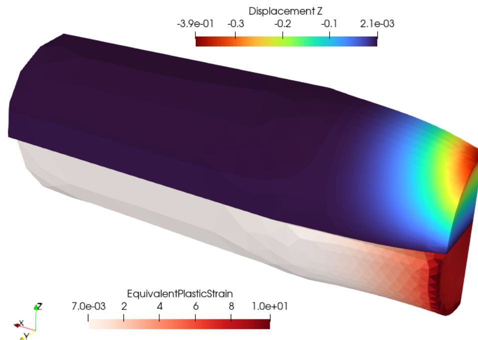



```python
E, nu = 70e3, 0.3
elastic_model = LinearElasticIsotropic(E, nu)

sig0 = 350.0
sigu = 500.0
b = 1e3
def yield_stress(p): # Voce-type exponential hardening
    return sig0 + (sigu - sig0) * (1 - jnp.exp(-b * p))

material = vonMisesIsotropicHardening(elastic_model,
     yield_stress)
```



$F^e F^p$ finite-strain plasticity

DEMO



Time spent	Unrolled AD	Implicit AD
Constitutive law	835 s	79 s
Linear solver	460 s	282 s

Material model calibration

Material behavior: $\boldsymbol{\sigma} = \mathsf{F}(\boldsymbol{\varepsilon}, \mathcal{S}_n; \boldsymbol{\theta})$ with material parameters $\boldsymbol{\theta}$
e.g. $\boldsymbol{\theta} = (E, \nu, \sigma_0, \sigma_u, b)$ isotropic elasticity + von Mises Voce hardening plasticity

Material model calibration

Material behavior: $\boldsymbol{\sigma} = \mathsf{F}(\boldsymbol{\varepsilon}, \mathcal{S}_n; \boldsymbol{\theta})$ with material parameters $\boldsymbol{\theta}$
e.g. $\boldsymbol{\theta} = (E, \nu, \sigma_0, \sigma_u, b)$ isotropic elasticity + von Mises Voce hardening plasticity

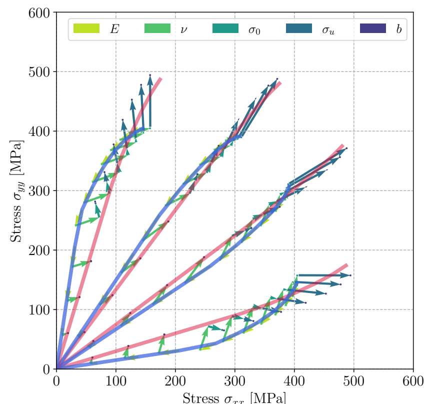
Calibration:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_k \|\boldsymbol{\sigma}^{(k)} - \boldsymbol{\sigma}_{\text{data}}^{(k)}\|^2$$

gradient-based optimisation, needs **material parameters sensitivities**

$$\frac{\partial \boldsymbol{\sigma}^{(k)}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathsf{F}}{\partial \boldsymbol{\theta}}(\boldsymbol{\varepsilon}^{(k)}, \mathcal{S}_n^{(k)}; \boldsymbol{\theta})$$

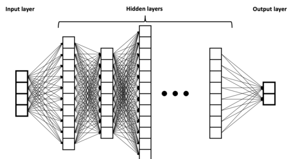
easy to obtain with **JAX**



Outline

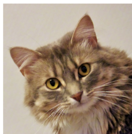
1. Automating PDEs with FEniCSx
2. Code generation for material constitutive modeling
3. JAX and Automatic Differentiation
4. **PDE-based optimisation**
   - Adjoint PDEs
   - Conic programming for non-smooth optimization

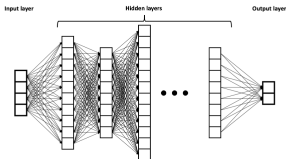
Introduction



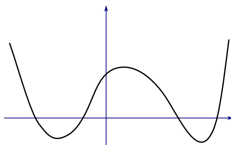
Wikimedia commons, Pixabay, Freepik

Introduction



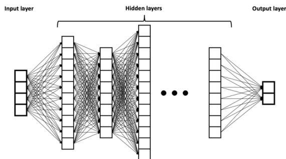


optimization is at the core of many fields of applications

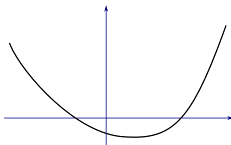


Wikimedia commons, Pixabay, Freepik

Introduction

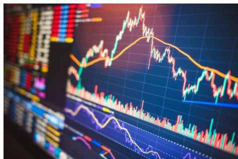


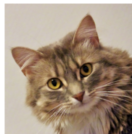
convex optimization is at the core of **many** fields of applications

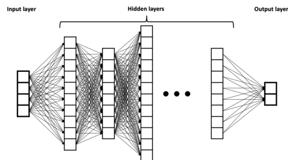


Wikimedia commons, Pixabay, Freepik

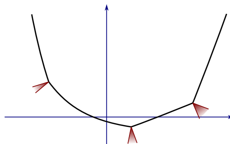
Introduction







non-smooth convex optimization is at the core of **many** fields of applications



Wikimedia commons, Pixabay, Freepik

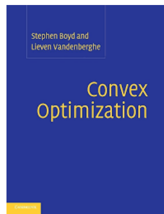
But why ?

- **guaranteed optimality** : local = global minima
- **algorithms efficiency** : often polynomial time complexity ⇒ **scalability**
- **mathematical elegance** of convex analysis (functions/sets/cones, duality, transforms, etc.) [Rockafellar, Moreau, 1970]
- **modeling power** : composition rules that preserve convexity
- **relaxation** of non-convex problems sometimes work very well, **reformulation** (convexity must be looked for when possible)

But why ?

- **guaranteed optimality** : local = global minima
- **algorithms efficiency** : often polynomial time complexity ⇒ **scalability**
- **mathematical elegance** of convex analysis (functions/sets/cones, duality, transforms, etc.) [Rockafellar, Moreau, 1970]
- **modeling power** : composition rules that preserve convexity
- **relaxation** of non-convex problems sometimes work very well, **reformulation** (convexity must be looked for when possible)
- **widespread tools** : open-source & commercial solvers (`Mosek`, `Gurobi`, `KNitro`) and modeling languages (`cvxpy`, `AMPL`), tutorials
- very good **books** and **lectures** [Boyd & Vandenberghe, Convex Optimization]

```python
import cvxpy as cp

w = cp.Variable(n)
gamma = cp.Parameter(nonneg=True)
ret = mu.T @ w
risk = cp.quad_form(w, Sigma)
prob = cp.Problem(cp.Maximize(ret - gamma * risk),
                  [cp.sum(w) == 1, w >= 0])
prob.solve()
```



Convex variational problems

variational equality:

$$\inf_{u \in V} J(u)$$

optimality conditions: $D_u J(u, v) = 0 \quad \forall v \in V$

Convex variational problems

variational inequalities arise in presence of contact, unilateral conditions, plasticity...

$$\begin{array}{ll} \inf_{u\in V} & J(u) \\ \text{s.t.} & u \in \mathcal{K} \end{array}$$

J convex function, $\mathcal{K}$ convex set

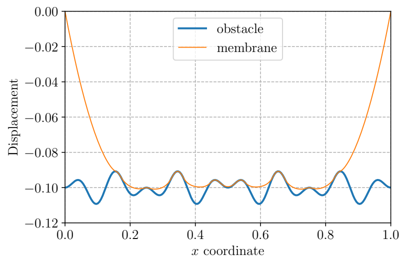
Convex variational problems

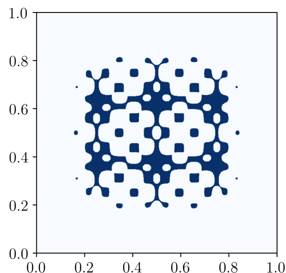
variational inequalities arise in presence of contact, unilateral conditions, plasticity...

$$\begin{array}{ll} \inf_{u\in V} & J(u) \\ \text{s.t.} & u \in \mathcal{K} \end{array}$$

J convex function, $\mathcal{K}$ convex set
e.g. **obstacle problem**:

$$\begin{array}{ll} \inf_{u\in V} & \displaystyle\int_\Omega \frac{1}{2}\|\nabla u\|_2^2 \,\mathrm{d}\Omega - \int_\Omega f u \,\mathrm{d}\Omega \\ \text{s.t.} & u \geq g \text{ on } \Omega \end{array}$$





Adjoint-based optimization

Adjoints are key ingredients for sensitivity analysis, optimal control, etc.

$$\text{Find } u \in V \text{ s.t. } F(u; m) = 0$$

- u is the PDE **solution**
- m is a **parameter**

Adjoint-based optimization

Adjoints are key ingredients for sensitivity analysis, optimal control, etc.

$$\text{Find } u \in V \text{ s.t. } F(u; m) = 0$$

- u is the PDE **solution**
- m is a **parameter**

Data assimilation: assume that we have a solution measurement u_{meas}
$\Rightarrow$ **Goal**: find m which best approximates u_{meas} e.g.

$$\begin{aligned} \min_m \quad & \int_\Omega \|u - u_{\text{meas}}\|^2 \,\mathrm{d}\Omega = J(m, u) \\ \text{s.t.} \quad & F(u; m) = 0 \end{aligned}$$

Adjoint-based optimization

Adjoints are key ingredients for sensitivity analysis, optimal control, etc.

$$\text{Find } u \in V \text{ s.t. } F(u; m) = 0$$

- u is the PDE **solution**
- m is a **parameter**

Data assimilation: assume that we have a solution measurement u_{meas}
$\Rightarrow$ **Goal**: find m which best approximates u_{meas} e.g.

$$\begin{aligned} \min_m \quad & \int_\Omega \|u - u_{\text{meas}}\|^2 \, d\Omega = J(m, u) \\ \text{s.t.} \quad & F(u; m) = 0 \end{aligned}$$

Reduced functional $R(m) = J(m; u(m)) \in \mathbb{R}$
$\Rightarrow$ optimization wrt m usually requires a **gradient descent**

How do we compute the **sensitivity** $\dfrac{dR}{dm}$?

Computing sensitivities

Tangent linear mode: $\dfrac{dR}{dm} = \dfrac{\partial J}{\partial m} + \dfrac{\partial J}{\partial u}\dfrac{du}{dm}$

Computing sensitivities

Tangent linear mode: $\dfrac{dR}{dm} = \dfrac{\partial J}{\partial m} + \dfrac{\partial J}{\partial u}\dfrac{du}{dm}$

We use the implicit function theorem to compute du/dm:

$$F(u(m); m) = 0 \quad \Rightarrow \quad \frac{\partial F}{\partial u}\frac{du}{dm} + \frac{\partial F}{\partial m} = 0$$

$$\frac{du}{dm} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m} \tag{2}$$

Computing sensitivities

Tangent linear mode: $\dfrac{dR}{dm} = \dfrac{\partial J}{\partial m} + \dfrac{\partial J}{\partial u}\dfrac{du}{dm}$

We use the implicit function theorem to compute du/dm:

$$F(u(m);m) = 0 \quad \Rightarrow \quad \frac{\partial F}{\partial u}\frac{du}{dm} + \frac{\partial F}{\partial m} = 0$$

$$\frac{du}{dm} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m} \tag{2}$$

for $m \in \mathbb{R}^k$, computing $\dfrac{du}{dm}$ requires solving k **linear systems**

Computing sensitivities

Tangent linear mode: $\dfrac{dR}{dm} = \dfrac{\partial J}{\partial m} + \dfrac{\partial J}{\partial u}\dfrac{du}{dm}$
We use the implicit function theorem to compute du/dm:

$$F(u(m); m) = 0 \quad \Rightarrow \quad \frac{\partial F}{\partial u}\frac{du}{dm} + \frac{\partial F}{\partial m} = 0$$

$$\frac{du}{dm} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m} \tag{2}$$

for $m \in \mathbb{R}^k$, computing $\dfrac{du}{dm}$ requires solving k **linear systems**
Adjoint mode: we take the adjoint:

$$\frac{dR^*}{dm} = \frac{\partial J^*}{\partial m} - \frac{\partial F^*}{\partial m}\left(\frac{\partial F^*}{\partial u}\right)^{-1}\frac{\partial J^*}{\partial u}$$

Computing sensitivities

Tangent linear mode: $\dfrac{dR}{dm} = \dfrac{\partial J}{\partial m} + \dfrac{\partial J}{\partial u}\dfrac{du}{dm}$

We use the implicit function theorem to compute du/dm:

$$F(u(m); m) = 0 \quad \Rightarrow \quad \frac{\partial F}{\partial u}\frac{du}{dm} + \frac{\partial F}{\partial m} = 0$$

$$\frac{du}{dm} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m} \tag{2}$$

for $m \in \mathbb{R}^k$, computing $\dfrac{du}{dm}$ requires solving k **linear systems**

Adjoint mode: we take the adjoint:

$$\frac{dR^*}{dm} = \frac{\partial J^*}{\partial m} - \frac{\partial F^*}{\partial m}\left(\frac{\partial F^*}{\partial u}\right)^{-1}\frac{\partial J^*}{\partial u}$$

Let p be the adjoint variable which solves:

$$p = \left(\frac{\partial F^*}{\partial u}\right)^{-1}\frac{\partial J^*}{\partial u} \tag{3}$$

$$\boxed{\frac{dR^*}{dm} = \frac{\partial J^*}{\partial m} - \frac{\partial F^*}{\partial m}p}$$

Computing sensitivities

Tangent linear mode: $\dfrac{dR}{dm} = \dfrac{\partial J}{\partial m} + \dfrac{\partial J}{\partial u}\dfrac{du}{dm}$
We use the implicit function theorem to compute du/dm:

$$F(u(m);m) = 0 \quad \Rightarrow \quad \frac{\partial F}{\partial u}\frac{du}{dm} + \frac{\partial F}{\partial m} = 0$$

$$\frac{du}{dm} = -\left(\frac{\partial F}{\partial u}\right)^{-1}\frac{\partial F}{\partial m} \tag{2}$$

for $m \in \mathbb{R}^k$, computing $\dfrac{du}{dm}$ requires solving k **linear systems**
Adjoint mode: we take the adjoint:

$$\frac{dR^*}{dm} = \frac{\partial J^*}{\partial m} - \frac{\partial F^*}{\partial m}\left(\frac{\partial F^*}{\partial u}\right)^{-1}\frac{\partial J^*}{\partial u}$$

Let p be the adjoint variable which solves:

$$p = \left(\frac{\partial F^*}{\partial u}\right)^{-1}\frac{\partial J^*}{\partial u} \tag{3}$$

$$\boxed{\frac{dR^*}{dm} = \frac{\partial J^*}{\partial m} - \frac{\partial F^*}{\partial m}p}$$

requires solving only **1 linear system** forward/reverse mode in Automatic Differentiation

Non-smooth optimization as conic programming

Linear programming

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^\mathsf{T}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned}$$

Non-smooth optimization as conic programming

Conic programming

$$\begin{array}{ll} \min_{\boldsymbol{x}} & \boldsymbol{c}^\mathsf{T}\boldsymbol{x} \\ \text{s.t.} & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \in \mathcal{K} \end{array}$$

Non-smooth optimization as conic programming

Conic programming

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^\mathsf{T}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \in \mathcal{K} \end{aligned}$$

where $\mathcal{K}$ is a product of elementary cones e.g.:

- positive orthants $\mathbb{R}^m_+$;
- Lorentz quadratic cones: $\mathcal{Q}_m = \{\mathsf{z} = (z_0, \bar{\mathsf{z}}) \in \mathbb{R}^+ \times \mathbb{R}^{m-1} \text{ s.t. } \|\bar{\mathsf{z}}\|_2 \leq z_0\}$
- semi-definite cones $\mathcal{S}^+_m$, the cone of semi-definite positive $m \times m$ symmetric matrices;
- power cones, exponential cones, etc.

Non-smooth optimization as conic programming

Conic programming

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \boldsymbol{c}^\mathsf{T}\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \in \mathcal{K} \end{aligned}$$

where $\mathcal{K}$ is a product of elementary cones e.g.:

- positive orthants $\mathbb{R}^m_+$;
- Lorentz quadratic cones: $\mathcal{Q}_m = \{\mathrm{z} = (z_0, \bar{\mathrm{z}}) \in \mathbb{R}^+ \times \mathbb{R}^{m-1} \text{ s.t. } \|\bar{\mathrm{z}}\|_2 \leq z_0\}$
- semi-definite cones $\mathcal{S}^+_m$, the cone of semi-definite positive $m \times m$ symmetric matrices;
- power cones, exponential cones, etc.

Solvers

interior-point algorithms, very efficient and robust (20-30 iterations)

The magic cone family

very large modelling power of **convex** functions and constraints

Convex functions and conic representation

Usual convex functions:

- $f(\boldsymbol{x}) = \boldsymbol{c}^\mathsf{T}\boldsymbol{x}$
- $f(\boldsymbol{x}) = \dfrac{1}{2}\boldsymbol{x}^\mathsf{T}\boldsymbol{Q}\boldsymbol{x}$ with $\boldsymbol{Q} \succeq 0$
- $f(\boldsymbol{x}) = \|\boldsymbol{x}\|_p$ with $p \geq 1$
- $f(\boldsymbol{x}) = \delta_G(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \in G \\ +\infty & \text{otherwise} \end{cases}$

  with G a convex set

Convex functions and conic representation

Usual convex functions:

- $f(\boldsymbol{x}) = \boldsymbol{c}^\mathsf{T}\boldsymbol{x}$
- $f(\boldsymbol{x}) = \dfrac{1}{2}\boldsymbol{x}^\mathsf{T}\boldsymbol{Q}\boldsymbol{x}$ with $\boldsymbol{Q} \succeq 0$
- $f(\boldsymbol{x}) = \|\boldsymbol{x}\|_p$ with $p \geq 1$
- $f(\boldsymbol{x}) = \delta_G(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \in G \\ +\infty & \text{otherwise} \end{cases}$ with G a convex set

Less usual convex functions:

- $f(\boldsymbol{X}) = \lambda_{\max}(\boldsymbol{X})$
- $f(\boldsymbol{x}, t) = \|\boldsymbol{x}\|^2/t$ with $t > 0$ (`QuadOverLin`)
- $f(\boldsymbol{x}) = \sum_{i=1}^k x_{[i]}$ with $x_{[i]}$ sorted entries in decreasing order (`top`$-k$)
- $f(\boldsymbol{x}) = \log\left(\sum_i \exp(x_i)\right)$ (`soft-max`)
- $f(\boldsymbol{X}) = \log\det(\boldsymbol{X}^{-1})$

Convex functions and conic representation

Usual convex functions:

- $f(\boldsymbol{x}) = \boldsymbol{c}^\mathsf{T}\boldsymbol{x}$
- $f(\boldsymbol{x}) = \dfrac{1}{2}\boldsymbol{x}^\mathsf{T}\boldsymbol{Q}\boldsymbol{x}$ with $\boldsymbol{Q} \succeq 0$
- $f(\boldsymbol{x}) = \|\boldsymbol{x}\|_p$ with $p \geq 1$
- $f(\boldsymbol{x}) = \delta_G(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \in G \\ +\infty & \text{otherwise} \end{cases}$
  with G a convex set

Less usual convex functions:

- $f(\boldsymbol{X}) = \lambda_{\max}(\boldsymbol{X})$
- $f(\boldsymbol{x}, t) = \|\boldsymbol{x}\|^2/t$ with $t > 0$ (`QuadOverLin`)
- $f(\boldsymbol{x}) = \sum_{i=1}^k x_{[i]}$ with $x_{[i]}$ sorted entries in decreasing order (`top`$-k$)
- $f(\boldsymbol{x}) = \log\left(\sum_i \exp(x_i)\right)$ (`soft-max`)
- $f(\boldsymbol{X}) = \log\det(\boldsymbol{X}^{-1})$

Convexity-preserving operations:

- sum
- supremum
- partial minimization
- Legendre-Fenchel transform
- inf-convolution
- perspective
- ...

Convex functions and conic representation

Usual convex functions:

- $f(\boldsymbol{x}) = \boldsymbol{c}^\mathsf{T}\boldsymbol{x}$
- $f(\boldsymbol{x}) = \dfrac{1}{2}\boldsymbol{x}^\mathsf{T}\boldsymbol{Q}\boldsymbol{x}$ with $\boldsymbol{Q} \succeq 0$
- $f(\boldsymbol{x}) = \|\boldsymbol{x}\|_p$ with $p \geq 1$
- $f(\boldsymbol{x}) = \delta_G(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \in G \\ +\infty & \text{otherwise} \end{cases}$
  with G a convex set

Convexity-preserving operations:

- sum
- supremum
- partial minimization
- Legendre-Fenchel transform
- inf-convolution
- perspective
- ...

Less usual convex functions:

- $f(\boldsymbol{X}) = \lambda_{\max}(\boldsymbol{X})$
- $f(\boldsymbol{x}, t) = \|\boldsymbol{x}\|^2/t$ with $t > 0$ (`QuadOverLin`)
- $f(\boldsymbol{x}) = \sum_{i=1}^k x_{[i]}$ with $x_{[i]}$ sorted entries in decreasing order (`top`$-k$)
- $f(\boldsymbol{x}) = \log\left(\sum_i \exp(x_i)\right)$ (`soft-max`)
- $f(\boldsymbol{X}) = \log\det(\boldsymbol{X}^{-1})$

Conic representation [Nesterov & Nemirovski]

$$\begin{aligned} f(\boldsymbol{x}) = \min_{\boldsymbol{y}} \quad & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{b}_l \leq \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b}_u \\ & \boldsymbol{y} \in \mathcal{K}_1 \times \ldots \times \mathcal{K}_p \end{aligned}$$

Convex functions and conic representation

Usual convex functions:

- $f(\boldsymbol{x}) = \boldsymbol{c}^\mathsf{T}\boldsymbol{x}$
- $f(\boldsymbol{x}) = \dfrac{1}{2}\boldsymbol{x}^\mathsf{T}\boldsymbol{Q}\boldsymbol{x}$ with $\boldsymbol{Q} \succeq 0$
- $f(\boldsymbol{x}) = \|\boldsymbol{x}\|_p$ with $p \geq 1$
- $f(\boldsymbol{x}) = \delta_G(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \in G \\ +\infty & \text{otherwise} \end{cases}$

  with G a convex set

Less usual convex functions:

- $f(\boldsymbol{X}) = \lambda_{\max}(\boldsymbol{X})$
- $f(\boldsymbol{x}, t) = \|\boldsymbol{x}\|^2/t$ with $t > 0$ (`QuadOverLin`)
- $f(\boldsymbol{x}) = \sum_{i=1}^k x_{[i]}$ with $x_{[i]}$ sorted entries in decreasing order (`top`$-k$)
- $f(\boldsymbol{x}) = \log\left(\sum_i \exp(x_i)\right)$ (`soft-max`)
- $f(\boldsymbol{X}) = \log\det(\boldsymbol{X}^{-1})$

Convexity-preserving operations:

- sum
- supremum
- partial minimization
- Legendre-Fenchel transform
- inf-convolution
- perspective
- ...

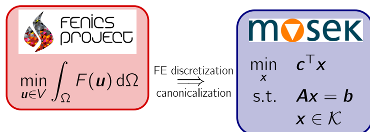
Conic representation [Nesterov & Nemirovski]

$$\begin{aligned} f(\boldsymbol{x}) = \min_{\boldsymbol{y}} \quad & \boldsymbol{c}^T\boldsymbol{x} + \boldsymbol{d}^T\boldsymbol{y} \\ \text{s.t.} \quad & \boldsymbol{b}_l \leq \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y} \leq \boldsymbol{b}_u \\ & \boldsymbol{y} \in \mathcal{K}_1 \times \ldots \times \mathcal{K}_p \end{aligned}$$

all mentioned operations preserve conic structures and can be explicitly computed

The dolfinx_optim package

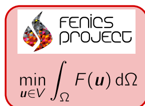
https://bleyerj.github.io/dolfinx_optim/



- **Domain-Specific Language** based on UFL for convex functions and their composition
- `Mosek` interior-point solver
- pre-defined **convex primitives**
  - `AbsValue`, `LinearTerm`, `QuadraticTerm`, `QuadOverLin`, etc.
  - vectors: `L1Norm`, `L2Norm`, `LinfNorm`, `LpNorm`, etc.
  - matrices: `SpectralNorm`, `NuclearNorm`, `FroebeniusNorm`, `LambdaMax`, etc.
- **composability** through convex-preserving transformations

The `dolfinx_optim` package

https://bleyerj.github.io/dolfinx_optim/



FE discretization $\Longrightarrow$ canonicalization

- **Domain-Specific Language** based on UFL for convex functions and their composition
- `Mosek` interior-point solver
- pre-defined **convex primitives**
  - `AbsValue`, `LinearTerm`, `QuadraticTerm`, `QuadOverLin`, etc.
  - vectors: `L1Norm`, `L2Norm`, `LinfNorm`, `LpNorm`, etc.
  - matrices: `SpectralNorm`, `NuclearNorm`, `FroebeniusNorm`, `LambdaMax`, etc.
- **composability** through convex-preserving transformations

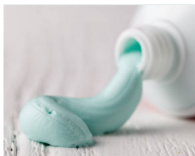
```
prob = MosekProblem(domain, name="Obstacle problem")
u = prob.add_var(V, bc=bc, lx=g)

prob.add_obj_func(-ufl.dot(f, u) * ufl.dx)

J = QuadraticTerm(ufl.grad(u), degree)
prob.add_convex_term(J)

prob.optimize()
```

Obstacle problem

$$\begin{aligned} \inf_{u\in V} \quad & \int_\Omega \frac{1}{2}\|\nabla u\|_2^2\,d\Omega - \int_\Omega fu\,d\Omega \\ \text{s.t.} \quad & u \geq g \text{ on } \Omega \end{aligned}$$

Viscoplastic fluids around us

cosmetics

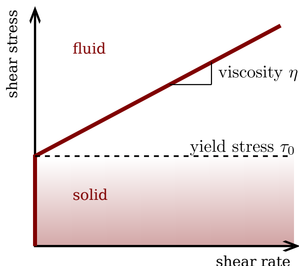


food

construction, geophysics

Formulation

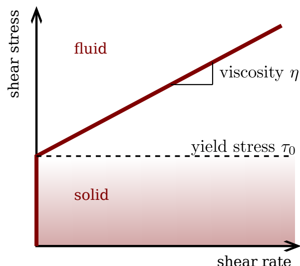
Viscoplastic fluids = a specific class of **non-Newtonian fluids** with a solid-like behaviour



- flow like a simple fluid above a **critical stress**
- remains at rest, like a solid, below

Formulation

Viscoplastic fluids = a specific class of **non-Newtonian fluids** with a solid-like behaviour



- flow like a simple fluid above a **critical stress**
- remains at rest, like a solid, below

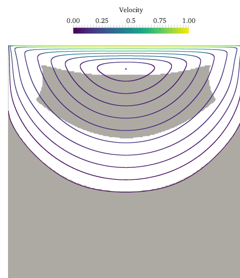
Primal variational principle: **smooth** + **non-smooth** term

$$\begin{array}{rl} \min\limits_{\boldsymbol{u},\boldsymbol{d}} & \displaystyle\int_\Omega \left(\frac{\eta}{2}\|\boldsymbol{d}\|^2 + \sqrt{2}\tau_0\|\boldsymbol{d}\|\right) \,\mathrm{d}\Omega - \int_\Omega \boldsymbol{f}\cdot\boldsymbol{u}\,\mathrm{d}\Omega \\ \text{s.t.} & \boldsymbol{d} = \frac{1}{2}(\nabla\boldsymbol{u} + \nabla^\mathsf{T}\boldsymbol{u}) \\ & \operatorname{div}\boldsymbol{u} = 0 \end{array}$$

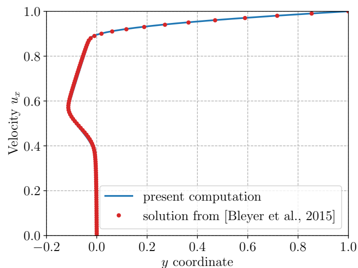
Viscoplastic fluid implementation

```
prob = MosekProblem(domain, "Viscoplastic fluid")

u = prob.add_var(V, bc=bc)

# mass conservation condition
Vp = fem.functionspace(domain, ("P", 1))
p = ufl.TestFunction(Vp)
prob.add_eq_constraint(p * ufl.div(u) * ufl.dx)

def strain(v):
    D = ufl.sym(ufl.grad(v))
    return ufl.as_vector([D[0, 0], D[1, 1], ufl.sqrt(2) * D[0, 1]])

visc = QuadraticTerm(strain(u), 2)
plast = L2Norm(strain(u), 2)

# add viscous term mu*||strain||_2^2
prob.add_convex_term(2 * mu * visc)
# add plastic term sqrt(2)*tau0*||strain||_2
prob.add_convex_term(np.sqrt(2) * tau0 * plast)

prob.optimize()
```

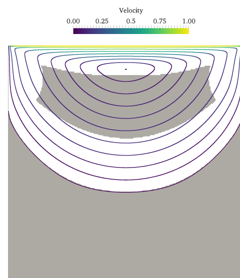
Viscoplastic fluid implementation



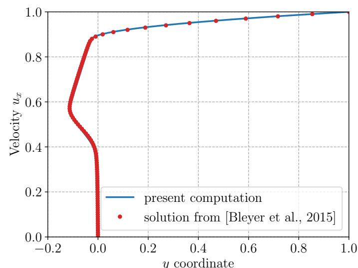
$\texttt{Bi} = 20$

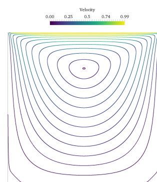


Viscoplastic fluid implementation

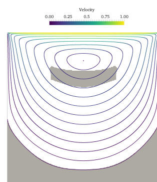


Bi = 20

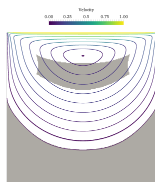




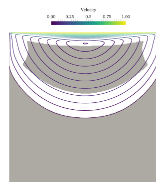
Bi = 0



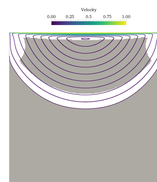
Bi = 2



Bi = 5



Bi = 50



Bi = 200

Variational cartoon/texture decomposition

Image $y = u$ (cartoon) + v (texture)

Y.Meyer's model (TV + G-norm) [Meyer, 2001]:

$$\begin{array}{ll} \inf\limits_{u,v} & \displaystyle\int_\Omega \|\nabla u\|_2 \,\mathrm{d}\Omega + \alpha \|v\|_G \\ \text{s.t.} & y = u + v \end{array}$$

$$\text{where} \quad \|v\|_G = \inf_{\boldsymbol{g}\in L^\infty(\Omega;\mathbb{R}^2)} \{\|\sqrt{g_1^2+g_2^2}\|_\infty \text{ s.t. } v = \operatorname{div} \boldsymbol{g}\}$$

reformulated as [Weiss et al., 2009]:

$$\begin{array}{ll} \inf\limits_{u,\boldsymbol{g}} & \displaystyle\int_\Omega \|\nabla u\|_2 \,\mathrm{d}\Omega \\ \text{s.t.} & y = u + \operatorname{div}(\boldsymbol{g}) \\ & \|\sqrt{g_1^2+g_2^2}\|_\infty \leq \alpha \end{array}$$

L_2 ad $L_{\infty,2}$-norms are **conic-representable** $\Rightarrow$ SOCP problem

Variational cartoon/texture decomposition

Image y : represented by a DG0 field on a 512x512 finite-element mesh
$u, \boldsymbol{g} \in \mathtt{CR} \times \mathtt{RT}$

```
prob = MosekProblem(domain, "Cartoon/texture decomposition")
Vu = fem.functionspace(domain, ("CR", 1))
Vg = fem.functionspace(domain, ("RT", 1))

u, g = prob.add_var([Vu, Vg], name=["Cartoon", "Texture"])


lamb_ = ufl.TestFunction(Vu)
constraint = ufl.dot(lamb_, u + ufl.div(g)) * ufl.dx
rhs = ufl.dot(lamb_, y) * ufl.dx
prob.add_eq_constraint(constraint, b=rhs)

tv_norm = L2Norm(ufl.grad(u), 0)
prob.add_convex_term(tv_norm)

g_norm = L2Ball(g / alpha, 2)
prob.add_convex_term(g_norm)

prob.optimize()
```

Variational cartoon/texture decomposition

Image y : represented by a DG0 field on a 512x512 finite-element mesh
$u, \boldsymbol{g} \in \mathtt{CR} \times \mathtt{RT}$

Original image

Cartoon layer

Texture layer

Barbara image

Limit analysis

Goal: find the maximum collapse load $F^+ = \lambda^+ F$ that a structure can sustain under a convex plasticity domain G

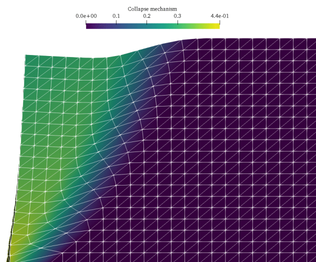
Plastic dissipation minimization principle:

$$\lambda^+ = \min_{\boldsymbol{u}\in\mathcal{U}_{\text{ad}}} \int_\Omega \pi_G(\boldsymbol{\varepsilon})\,\mathrm{d}\Omega \quad \text{s.t.} \quad \int_\Omega \boldsymbol{f}\cdot\boldsymbol{u}\,\mathrm{d}\Omega + \int_{\partial\Omega_\mathbf{N}} \boldsymbol{T}\cdot\boldsymbol{u}\,\mathrm{dS} = 1$$

$$\pi_G(\boldsymbol{\varepsilon}) = \sup_{\boldsymbol{\sigma}\in G} \boldsymbol{\sigma}:\boldsymbol{\varepsilon}$$

e.g. Mohr-Coulomb 3D criterion: $\pi_G(\boldsymbol{\varepsilon}) = \begin{cases} c\,\text{cotan}\,\phi\,\text{tr}\,\boldsymbol{\varepsilon} & \text{if } \text{tr}(\boldsymbol{\varepsilon}) \geq \sin\phi \sum_I |\varepsilon_I| \\ +\infty & \text{otherwise} \end{cases}$

```
class MohrCoulomb(ConvexTerm):
    """SDP implementation of Mohr-Coulomb criterion."""
    def conic_repr(self, X):
        Y1 = self.add_var((3,3), cone=SDP(3))
        Y2 = self.add_var((3,3), cone=SDP(3))
        a = (1 - ufl.sin(phi)) / (1 + ufl.sin(phi))
        self.add_eq_constraint(X - to_vect(Y1) + to_vect(Y2))
        self.add_eq_constraint(ufl.tr(Y2) - a * ufl.tr(Y1))
        self.add_linear_term(2 * c * ufl.cos(phi) / (1 + ufl.sin(phi))
       * ufl.tr(Y1))
```



To conclude...

Automated tools for scientific documentation

Welcome

Introduction

Linear elasticity

Hyperelasticity

Reissner-Mindlin plates

Tours

Linear problems

Isotropic and orthotropic plane stress elasticity ☆

Axisymmetric formulation for elastic structures of revolution ☆

Linear thermoelasticity (weak coupling) ☆

Thermo-elastic evolution problem (full coupling) ☆☆

Plates

Shear-locking in thick plate models with quadrilateral elements ☆

Shells

Linear shell model ☆☆

Generating a shell model with the `Gmsh` Python API ☆

Nonlinear problems

Elasto-plastic analysis of a 2D von Mises material ☆☆☆

Localized 1D solutions in phase-field approach to brittle fracture ☆

Numerical Tours of Computational Mechanics with FEniCSx

Jeremy Bleyer

Find $\boldsymbol{u} \in V$ such that:

$$\int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \nabla^s \boldsymbol{v} \,\mathrm{d}\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \,\mathrm{d}\Omega \quad \forall \boldsymbol{v} \in V$$

```
a = inner(sigma(u), sym(grad(v))) * dx
L = inner(f, v) * dx
```

Welcome

What is it about ?

These numerical tours will introduce you to a wide variety of topics in computational continuum and structural mechanics using the finite element software `FEniCSx`, http://fenicsproject.org.

This book is organized in the following different parts:

Introduction

This part contains a set of short numerical tours to get introduced to `FEniCSx` via basic examples:

- a linear elasticity problem

Contents

What is it about ?
Introduction
Tours
Tips & Tricks
Citing
About the author

- Versioning: `git`
- Docstrings and comments
- Automatic doc generation: Sphinx
- Demos: Jupyter Notebooks, Jupytext
- Publishing: JupyterBook, Readthedocs, Github Pages

comply very well with **open science** concepts