Automation of multiphysics numerical simulations using open-source software

Jeremy Bleyer
coll. Paul Bouteiller, Jørgen Dokken, Alice Gribonval, Jack S. Hale, Thomas Helfer, Andrey Latyshev, Corrado Maurini, Maxime Pierre

Ecole Nationale des Ponts et Chaussées, IP Paris

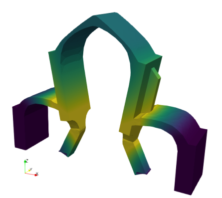
Funded by the European Union

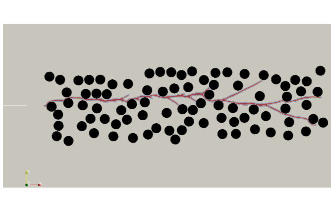
European Research Council
Established by the European Commission

January 22, 2026

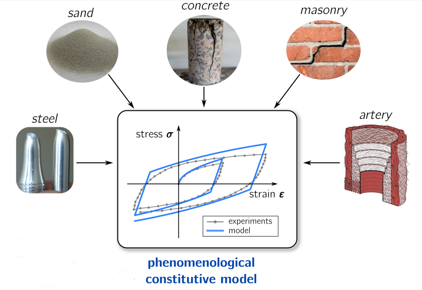
Context

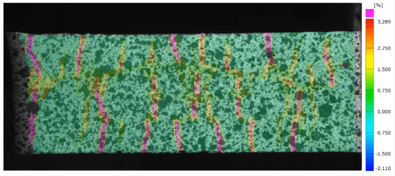
Structural simulations: numerical methods (FE), nonlinearities, complex geometry, time-dependency





Constitutive behavior: elasticity, viscoelasticity, plasticity, damage, temperature effects...





Open-source software for computational mechanics

Advantages/Drawbacks

reproducibility

extensibility

community

user responsibility

learning curve

stability

- code_aster
- deal_ii
- elmer
- Kratos

- Cast3m/Manta
- MOOSE
- MFEM

- OpenFOAM
- Basilisk
- FreeFEM++

- gmsh
- pyvista
- Paraview
- pytorch

Here: FEniCSx – MFront – JAX

FEniCSx

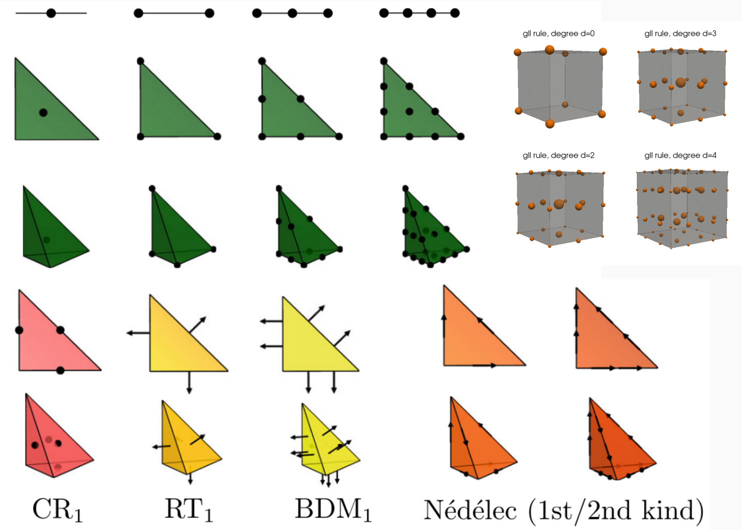
collection of free, open source, software components for **automated solution** of **Partial Differential Equations** (PDEs)

FEniCS PROJECT

Features:

- variational formulation expressed with **math symbolic language**: `ufl.grad`, `ufl.div`, `ufl.derivative`, etc.
- **code-generation** (`C`) and **JIT compilation** of FE kernels
- **extensive library** of finite elements
- designed for **parallel HPC computations**

Element types



A simple elasticity example

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

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

with **isotropic linear elasticity**: $\boldsymbol{\sigma}(\boldsymbol{u}) = \lambda \operatorname{tr}(\boldsymbol{\varepsilon})\boldsymbol{I} + 2\mu\boldsymbol{\varepsilon}$ with $\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u}$.

```
dim, degree = mesh.geometry.dim, 2
V = fem.functionspace(mesh, ("P", degree, dim))

def epsilon(v):
    return ufl.sym(ufl.grad(v))
def sigma(v):
    return lmbda * ufl.tr(epsilon(v)) * ufl.Identity(dim) + 2 * mu * epsilon(v)

u, v = TrialFunction(V), TestFunction(V)

a, L = ufl.inner(sigma(u), epsilon(v)) * dx, ufl.dot(f, v) * dx
```

Multi-field variational problems

One of the big advantages of **FEniCS**
e.g. **Stokes incompressible fluids**: mixed $u - p$ formulation

Taylor-Hood element (P_2/P_1)

```
# Define function space
P2 =  basix.ufl.element('P', tetrahedron, 2, shape=(dim,))
P1 =  basix.ufl.element('P', tetrahedron, 1)
THe = basix.ufl.mixed_element([P2, P1])
W = fem.functionspace(mesh, THe)
```

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

Multi-field variational problems

One of the big advantages of **FEniCS**
e.g. **Stokes incompressible fluids**: mixed $u - p$ formulation

MINI element $(P_1 + \mathcal{B})/P_1$

```
# Define function space
Pv1 = basix.ufl.element('P', tetrahedron, 1)
B = basix.ufl.element('Bubble', tetrahedron, 3)
Mini = basix.ufl.mixed_element([Pv1+B, P1])
W =  fem.functionspace(mesh, Mini)
```

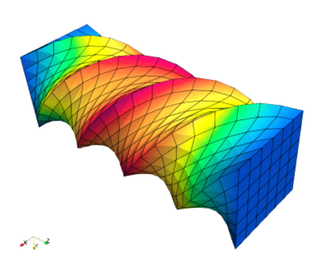
```
# Define variational problem
(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
a, L = inner(grad(u), grad(v))*dx - p*div(v)*dx + div(u)*q*dx, dot(f, v)*dx
```

Finite-strain mechanics

Finite-strain weak equilibrium

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

$\boldsymbol{P} = \dfrac{\partial \psi}{\partial \boldsymbol{F}}$: first Piola-Kirchhoff stress given by a **hyperelastic potential** $\psi(\boldsymbol{F})$ with $\boldsymbol{F} = \boldsymbol{I} + \nabla \boldsymbol{u}$.



Example: $\psi(\boldsymbol{F}) = \dfrac{\mu}{2}(\bar{I}_1 - 3) + k(J^2 + J^{-2} - 2)$ s.t. $\bar{I}_1 = \mathrm{tr}(\overline{\boldsymbol{C}})$, $\overline{\boldsymbol{C}} = J^{-2/3}\boldsymbol{F}^\mathsf{T}\boldsymbol{F}$, $J = \det \boldsymbol{F}$

Tangent operator

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

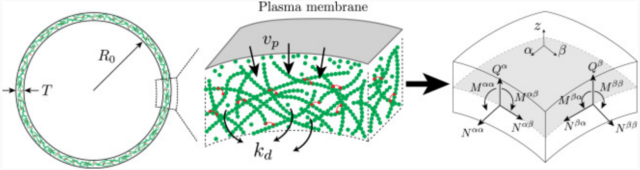
$\Rightarrow$ **UFL symbolic differentiation**

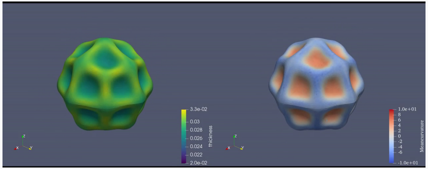
```python
def psi(u):
    ...

E_pot = psi * dx - ufl.dot(f, u)*dx
Res = ufl.derivative(E_pot, u, v)
Jac = ufl.derivative(Res, u, du)
```

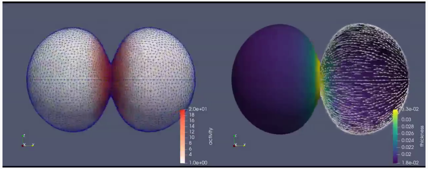
A biomechanics application [with Hudson Borja da Rocha and Hervé Turlier (Collège de France)]

Mechanics of biological cells

- The **cell cortex** is a very thin layer under the membrane that controls cell shape.
- It **actively generates forces and flows** (e.g. during cell division or deformation).
- We model its mechanics using an **active viscous shell model** [JMPS, 2022]







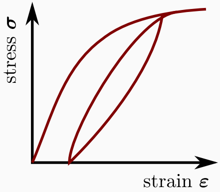
MFront

Material constitutive modeling

Constitutive model = a **stress-strain** relation

$$\boldsymbol{\varepsilon} \longrightarrow \boxed{\text{CONSTITUTIVE RELATION}} \longrightarrow \boldsymbol{\sigma}$$

generally with **custom implementation** in a FE solver

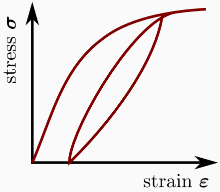


Material constitutive modeling

Constitutive model = a **stress-strain** relation

$$\boldsymbol{\varepsilon} \longrightarrow \boxed{\text{CONSTITUTIVE RELATION}} \longrightarrow \boldsymbol{\sigma}$$

generally with **custom implementation** in a FE solver

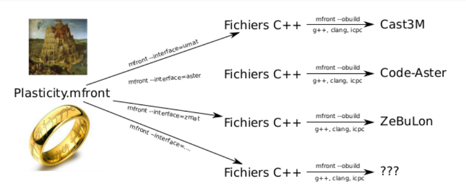


- **decouple** material library from structural solvers
- **inter-operability**: **material knowledge** can be **shared** across various solvers (industrial and academic)
- improve quality assurance, long-term sustainability, collaboration

MFront

`MFront` = a **code generation tool** dedicated to **material knowledge**
Main developer: Thomas Helfer (CEA Cadarache) + many other contributors

- Numerical **efficiency**
- **Inter-operability**: specific (`Cast3M`, `code_aster`, `Europlexus`, `Abaqus`, `Zebulon`, `Ansys`, etc.) and *generic* interfaces (`python`, `fortran`, `c`, `julia`, etc.)
- Ease of use (**Domain-Specific Languages**)



Features

- C++ code, text file input to be compiled for a given interface
- **Typical mechanical behaviors**: elasticity, viscoelasticity, plasticity, damage, temperature dependence, finite strains, cohesive zone models, *homogenized behaviors*
- **solvers**: implicit/explicit time integrators, (quasi)-Newton
- `mtest`: testing tool

Examples

Pre-defined modular bricks

```
@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},
}
};
```

Examples

Pre-defined modular bricks

```
@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},
}
};
```

```
@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
```

Advanced aspects

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

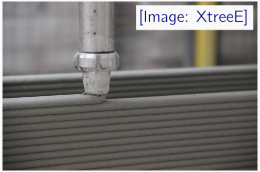
Mechanics

$$\Delta\boldsymbol{\varepsilon}_{n+1}, \boldsymbol{\sigma}_n, \boldsymbol{\alpha}_n \to \boxed{\texttt{MFront}} \to \boldsymbol{\sigma}_{n+1}, \boldsymbol{\alpha}_{n+1}, \frac{d\boldsymbol{\sigma}_{n+1}}{d\boldsymbol{\varepsilon}_{n+1}}$$

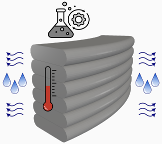
Multiphysics

$$(\Delta\boldsymbol{g}^1, \ldots, \Delta\boldsymbol{g}^p)_{n+1}, (\boldsymbol{\sigma}^1, \ldots, \boldsymbol{\sigma}^p)_n, \boldsymbol{\alpha}_n \to \boxed{\texttt{MFront}} \to (\boldsymbol{\sigma}^1, \ldots, \boldsymbol{\sigma}^p)_{n+1}, \boldsymbol{\alpha}_{n+1}, \frac{\partial \boldsymbol{\sigma}^i_{n+1}}{\partial \boldsymbol{g}^j_{n+1}}$$

Early age concrete = a **thermo-poro-chemo-mechanical** model [M. Pierre et al.]



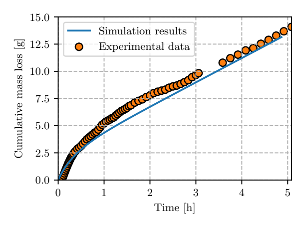
$$
\begin{cases}
\mathrm{d}\boldsymbol{\sigma} = \mathbb{C}(\xi) : \mathrm{d}\boldsymbol{\varepsilon} - b(\xi) S_\ell \mathrm{d}p \mathbf{1} - 3\alpha K(\xi) \mathrm{d}T \mathbf{1} \\
\mathrm{d}\phi = b(\xi) \mathrm{tr}(\mathrm{d}\boldsymbol{\varepsilon}) + \frac{b(\xi) - \phi_0(\xi)}{K_s} S_\ell \mathrm{d}p - 3\alpha \left(b(\xi) - \phi_0(\xi)\right) \mathrm{d}T - \Delta \overline{V}_s \mathrm{d}\xi \\
\mathrm{d}S_s = 3\alpha K(\xi) \mathrm{tr}(\mathrm{d}\boldsymbol{\varepsilon}) - 3\alpha \left(b(\xi) - \phi_0(\xi)\right) S_\ell \mathrm{d}p + \frac{C(1-\phi_0(\xi))}{T_0} \mathrm{d}T - \frac{\mathcal{L}}{T_0} \mathrm{d}\xi
\end{cases}
$$

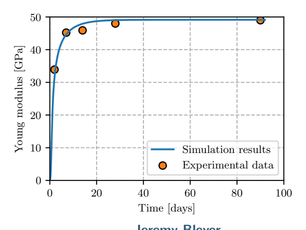


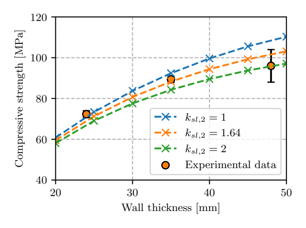
3D-printed concrete

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

[A. Gribonval et al.]: influence of **environmental conditions** on **compressive strength**

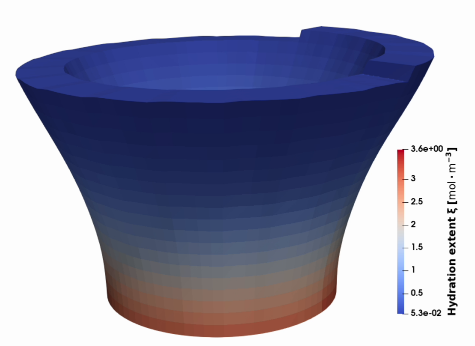






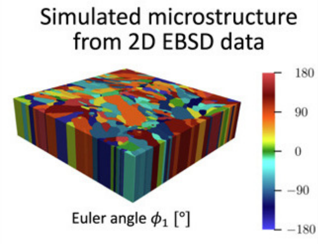
3D-printing concrete modeling

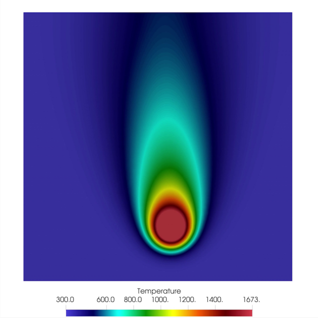
[M. Pierre]: `gmsh` - `FEniCSx` - `MFront` pipeline with mesh activation
THCM model with elastoplastic Cam-Clay behavior

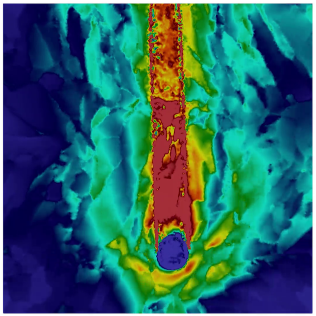


Residual stresses after laser scanning of additively manufactured stainless steel

coll. Nikhil Mohanan, Manas Upadhyay (LMS, Ecole Polytechnique)







MFront: thermo elastoviscoplastic polycrystal model
FEniCSx: weakly coupled thermomechanics on polycrystal mesh

JAX

Constitutive models as ML models

Constitutive models can be formalized as **parametrized functions** (stress update):

$$\boxed{\boldsymbol{\sigma}_{n+1}, \boldsymbol{\alpha}_{n+1} = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_{n+1}, \boldsymbol{\alpha}_n)}$$

where $\boldsymbol{\theta}$ are the model **material parameters** and $\mathcal{M}_\theta = \mathcal{O}_1 \circ \mathcal{O}_2 \circ \cdot \circ \mathcal{O}_n$

Constitutive models as ML models

Constitutive models can be formalized as **parametrized functions** (stress update):

$$\boxed{\boldsymbol{\sigma}_{n+1}, \boldsymbol{\alpha}_{n+1} = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_{n+1}, \boldsymbol{\alpha}_n)}$$

where $\boldsymbol{\theta}$ are the model **material parameters** and $\mathcal{M}_\theta = \mathcal{O}_1 \circ \mathcal{O}_2 \circ \cdot \circ \mathcal{O}_n$

Formally equivalent with ML, main **differences** are:

- **internal state** not **observable** from data
- **sequential evaluation of the model** in time:

$$\boldsymbol{\sigma}_1, \boldsymbol{\alpha}_1 = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_1, \boldsymbol{\alpha}_0) \to \boldsymbol{\sigma}_2, \boldsymbol{\alpha}_2 = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_2, \boldsymbol{\alpha}_1) \to \cdots \to \boldsymbol{\sigma}_{n+1}, \boldsymbol{\alpha}_{n+1} = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_n, \boldsymbol{\alpha}_n)$$

- some **elementary operations** $\mathcal{O}_i$ may not have **closed-form** expressions

Constitutive models as ML models

Constitutive models can be formalized as **parametrized functions** (stress update):

$$\boxed{\boldsymbol{\sigma}_{n+1}, \boldsymbol{\alpha}_{n+1} = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_{n+1}, \boldsymbol{\alpha}_n)}$$

where $\boldsymbol{\theta}$ are the model **material parameters** and $\mathcal{M}_\theta = \mathcal{O}_1 \circ \mathcal{O}_2 \circ \cdot \circ \mathcal{O}_n$

Formally equivalent with ML, main **differences** are:

- **internal state** not **observable** from data
- **sequential evaluation of the model** in time:

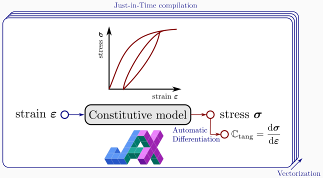
$$\boldsymbol{\sigma}_1, \boldsymbol{\alpha}_1 = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_1, \boldsymbol{\alpha}_0) \to \boldsymbol{\sigma}_2, \boldsymbol{\alpha}_2 = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_2, \boldsymbol{\alpha}_1) \to \cdots \to \boldsymbol{\sigma}_{n+1}, \boldsymbol{\alpha}_{n+1} = \mathcal{M}_{\boldsymbol{\theta}}(\boldsymbol{\varepsilon}_n, \boldsymbol{\alpha}_n)$$

- some **elementary operations** $\mathcal{O}_i$ may not have **closed-form** expressions

Opportunities

- **phenomenological** or **NN-based** models can be treated in a **unifying framework**
- they can be **seamlessly hybridized**!
- material parameter **identification** can benefit from **ML gradient-based algorithms**
- **practical implementation** can benefit from **modern ML frameworks**

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):
    [...]
    return stress, new_state
```

- **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)
```

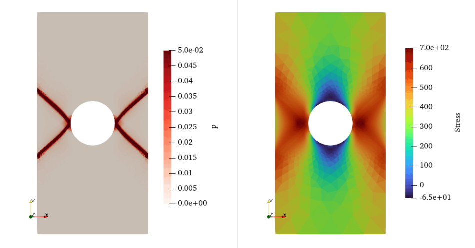
`jaxmat` – an open-source material modeling library

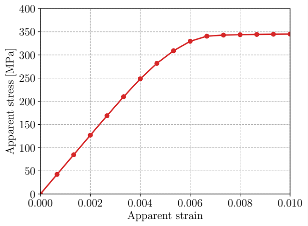
First release end of 2025: `https://github.com/bleyerj/jaxmat`

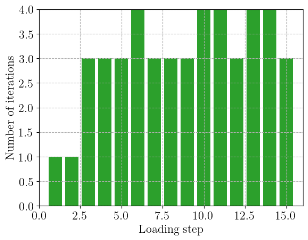
Features

- based on `JAX` and `equinox` (NN models), `optimistix` (optimization solvers), `diffrax` (ODE solvers)
- models as **PyTrees** $\Rightarrow$ allows **modularity** and **hierarchical composition**:
  $\boldsymbol{\theta} = \{\boldsymbol{\theta}_{\text{elastic}}, \{\boldsymbol{\theta}_{\text{plastic}}, \boldsymbol{\theta}_{\text{hardening}}\}\}$
- prototype **material models**: elastoplasticity, viscoplasticity, hyperelasticity, finite strain plasticity, viscoelasticity, Generalized Standard Materials, damage, etc.
- **hybrid NN-phenomenological** models
- **differentiable solvers** using **implicit differentiation**
- **gradient-based** identification
- **CPU** or **GPU** acceleration
- **compatibility** with FEniCSx (`v.0.10`)

Small-strain elastoplasticity








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

class VoceHardening(equinox.Module):
    sig0 = 350.0
    sigu = 500.0
    b = 1e3
    def __call__(self, p):
        return self.sig0 + (self.sigu - self.sig0) * (1 - jnp.exp(-self.b *
      p))

material = vonMisesIsotropicHardening(elastic_model,
    yield_stress)
```

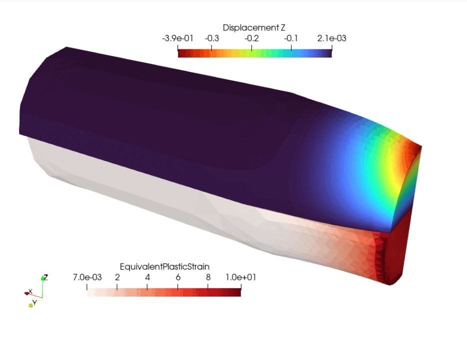
$\boldsymbol{F}^{\text{e}}\boldsymbol{F}^{\text{p}}$ finite-strain plasticity

$$\boldsymbol{F} = \boldsymbol{F}^{\text{e}}\boldsymbol{F}^{\text{p}}\,; \quad \bar{\boldsymbol{b}}^{\text{e}} = J^{-2/3}\boldsymbol{F}^{\text{e}}(\boldsymbol{F}^{\text{e}})^{\text{T}}$$

$$\boldsymbol{\tau} = \mu\,\text{dev}(\bar{\boldsymbol{b}}^{\text{e}}) + \frac{\kappa}{2}(J^2 - 1)\boldsymbol{I}$$

$$f(\bar{\boldsymbol{b}}^{\text{e}}) = \mu\|\boldsymbol{s}\| - \sqrt{\frac{2}{3}}R(p_n + \Delta p) \leq 0$$

$$0 = \text{dev}(\bar{\boldsymbol{b}}^{\text{e}} - \bar{\boldsymbol{b}}^{\text{e}}_{\text{trial}}) + \sqrt{\frac{2}{3}}\Delta p\,\text{tr}(\bar{\boldsymbol{b}}^{\text{e}})\frac{\boldsymbol{s}}{\|\boldsymbol{s}\|}$$

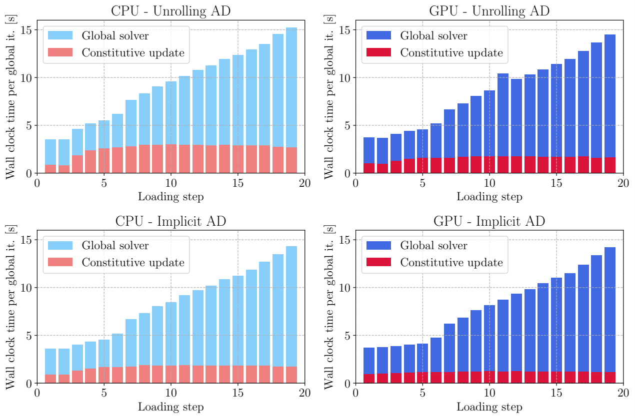


Resolution involves local solving of a Newton system of size 7
Tangent operator in $PK1/F$ using **implicit AD**:

$$\boldsymbol{P} = \boldsymbol{\tau}\boldsymbol{F}^{-1} \qquad \mathbb{C}_{\text{tang}} = \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}}$$

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

Constitutive equation: `jax[cpu]` or `jax[gpu]` on `NVIDIA RTX A1000`
Linear solver: PETSc `gmres` + `gamg`



Global linear solves – Constitutive behavior integration

Material model identification

Identification

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \frac{1}{M}\sum_{k=1}^{M} \|\boldsymbol{\sigma}^{(k)} - \boldsymbol{\sigma}_{\text{data}}^{(k)}\|^2$$

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

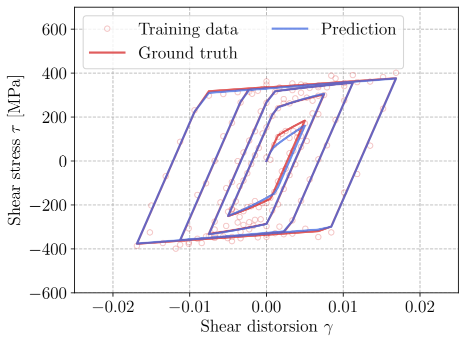
$$\frac{\partial \boldsymbol{\sigma}^{(k)}}{\partial \boldsymbol{\theta}} \to \frac{\partial \mathsf{F}}{\partial \boldsymbol{\theta}}(\boldsymbol{\varepsilon}, \boldsymbol{\alpha}; \boldsymbol{\theta})$$

easy to obtain with **JAX**

```
@jax.jit
def loss(material, sig_data, loading):
    sig_hat = compute_evolution(material, loading)
    loss = optax.l2_loss(sig_hat, sig_data)
    return loss.mean()
```

differentiable time loop with `jax.lax.scan`

Armstrong-Frederick type hardening plasticity with **noisy data**



Open source and industry

Undabit: an Italian vibro-acoustic company

Simvia: subsidiary of **EDF**

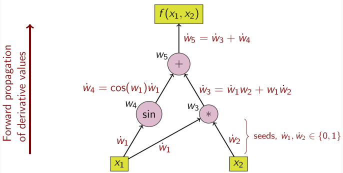
- training, technical support
- software projects, cloud & HPC

At Simvia, we believe that open-source [...] give industrial companies control over their simulation solutions, by offering them reactivity, performance, and scalability.

Thank you for your attention!

What is Automatic Differentiation ?

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

Return mapping algorithm

Elastic predictor $\boldsymbol{\sigma}_{\text{elas}} = \boldsymbol{\sigma}_n + \mathbb{C} : \Delta\boldsymbol{\varepsilon}$

$f_{\text{elast}} = \sigma_{\text{eq}} - R(p_n)$

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

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

implement **Solve** using fixed-point algorithm, Newton method, bisection, etc.

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

Differentiating through elastoplasticity

Return mapping algorithm

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

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

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

implement **Solve** using fixed-point algorithm, Newton method, bisection, etc.

Every step is differentiable with AD *a priori*, **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:

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

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 $[\partial_x F]$

Can be implemented for any function using `jax.lax.custom_root`
Available in `optimistix`:

```
solver = optimistix.Newton(rtol=1e-8, atol=1e-8)
solution = optimistix.root_find(fun, solver, y0, adjoint=optimistix.ImplicitAdjoint())
```

PANN models for data-driven hyperelasticity

Isotropic hyperelastic potential

$$\psi(\boldsymbol{F}) = \Psi(I_1, I_2, I_3)$$

polyconvexity guaranteed if Ψ is **convex**

Physics-Augmented Neural Networks [Linden et al., 2023] approximate

$$\psi_{\boldsymbol{\theta}}(\boldsymbol{F}) = \Psi_{\boldsymbol{\theta}}(I_1, I_2, I_3, -2J)$$

where $\Psi_{\boldsymbol{\theta}}(\mathcal{I})$ is an **Input Convex Neural Network** (ICNN) [Amos et al., 2017] (positive weights, `softplus` activation)

Important: enforce stress-free initial configuration

$$\Psi_{\boldsymbol{\theta}}^{\text{PANN}}(\boldsymbol{C}) = \Psi_{\boldsymbol{\theta}}(\mathcal{I}) - \left.\frac{\partial \Psi_{\boldsymbol{\theta}}}{\partial \mathcal{I}}\right|_{\boldsymbol{C}=\boldsymbol{I}} 2(J-1)$$

```
class PANN(ICNN):
    def nn_energy(self, lambC):
        I3 = jnp.prod(lambC)
        I1 = jnp.sum(lambC)
        I2 = jnp.sum(I3 / lambC)
        return super().__call__(jnp.asarray([I1, I2,
  I3, -2*jnp.sqrt(I3)]))
```

```
    def pann_energy(self, lambC):
        J = jnp.sqrt(jnp.prod(lambC))
        dpsi_dC =
  jax.jacfwd(self.nn_energy)(jnp.ones((3,)))[0]
        return self.nn_energy(lambC) - 2 * dpsi_dC *
  (J - 1)
```

PANN model on rubber elasticity data [Treloar, 1944]

PANN with 4 inputs, 10 neurons in 1 hidden layer