# Differentiable Constitutive Modeling with FEniCSx and JAX

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

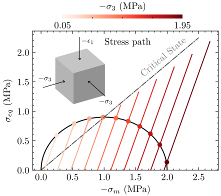
*Ecole Nationale des Ponts et Chaussées*
*Laboratoire Navier, ENPC, IP Paris, Univ Gustave Eiffel, CNRS*

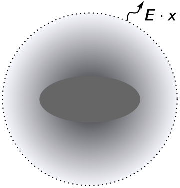
*FEniCS 2025*
June 18$^{\text{th}}$-20$^{\text{th}}$ 2025

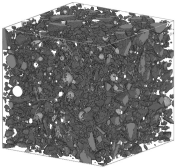
# Constitutive modeling

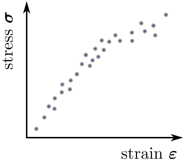
**Constitutive behavior**: complements **balance equations** and **kinematic relations**
e.g. elasticity, viscoelasticity, plasticity, damage, temperature effects...

- **Modelling approaches**: phenomenological, micromechanics/mean-field, computational ($FE^2$ , FFT,reduced models), data-driven
- **Thermodynamics**: path/history-dependence, internal state variables, evolution equations



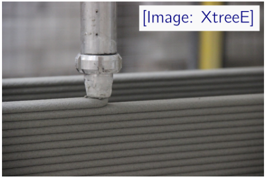


# Constitutive modeling

**Constitutive behavior**: complements **balance equations** and **kinematic relations**
e.g. elasticity, viscoelasticity, plasticity, damage, temperature effects...

- **Modelling approaches**: phenomenological, micromechanics/mean-field, computational ($FE^2$ , FFT,reduced models), data-driven
- **Thermodynamics**: path/history-dependence, internal state variables, evolution equations
- **Generalized continua**: strain gradient, Cosserat, micromorphic, internal length
- **Multi-physics**: strongly coupled behaviors e.g. poromechanics

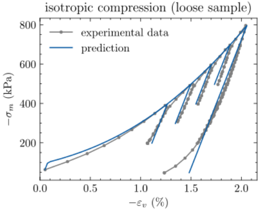
$$\begin{aligned}
d\boldsymbol{\sigma} &= \mathbb{C}_\xi : d\boldsymbol{\varepsilon} - b_\xi S_\ell \, dp \boldsymbol{I} - 3\alpha K_\xi \, dT \boldsymbol{I} \\
d\phi &= b_\xi \operatorname{tr}(d\boldsymbol{\varepsilon}) + \frac{b_\xi - \phi_{0\xi}}{K_s} dp - 3\alpha(b_\xi - \phi_{0\xi}) \, dT - \Delta V_s \, d\xi \\
dS_s &= 3\alpha K_\xi \operatorname{tr}(\boldsymbol{\varepsilon}) - 3\alpha(b_\xi - \phi_{0\xi}) \, dp + C \frac{1 - \phi_{0\xi}}{T_0} dT + \frac{\mathcal{L}_\xi}{T_0} d\xi
\end{aligned}$$

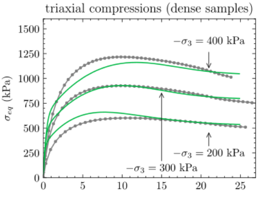


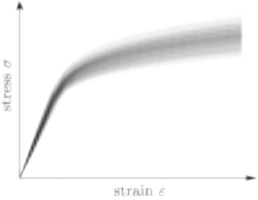
## Constitutive modeling

**Constitutive behavior**: complements **balance equations** and **kinematic relations** e.g. elasticity, viscoelasticity, plasticity, damage, temperature effects...

- **Modelling approaches**: phenomenological, micromechanics/mean-field, computational ($FE^2$ , FFT,reduced models), data-driven
- **Thermodynamics**: path/history-dependence, internal state variables, evolution equations
- **Generalized continua**: strain gradient, Cosserat, micromorphic, internal length
- **Multi-physics**: strongly coupled behaviors e.g. poromechanics
- **Material properties**: calibration/identification, variability/uncertainties



# Outline

## Computational aspects of constitutive modeling

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

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

**Local** non-linear (implicit) mapping

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

## Computational aspects of constitutive modeling

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

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

**Local** non-linear (implicit) mapping

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

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

## Computational aspects of constitutive modeling

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

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

**Local** non-linear (implicit) mapping

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

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

### Tangent operators

**Newton** method for solving (1) requires the **Jacobian**:

$$\text{e.g.} \qquad \delta\boldsymbol{\sigma}(\nabla^s \boldsymbol{u}) = \frac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{\varepsilon}} : \nabla^s \delta \boldsymbol{u}$$

sometimes also $\frac{\partial \boldsymbol{\sigma}}{\partial \mathcal{S}_n}$, $\frac{\partial \mathcal{S}_{n+1}}{\partial \boldsymbol{\varepsilon}}$, $\frac{\partial \mathcal{S}_{n+1}}{\partial \mathcal{S}_n}$ (multiphysics)

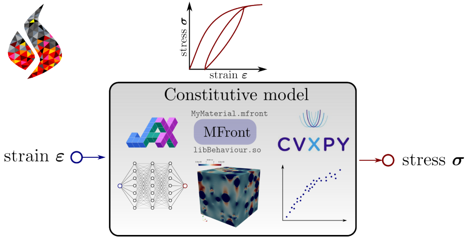
## dolfinx_materials: Python package for material behaviors

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

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

**Concrete implementation** of the constitutive relation

- a user-defined Python function
- provided by an external library (e.g. behaviors compiled with `MFront`,`UMAT`s, etc.)
- convex optimization solvers
- neural networks, model-free data-driven, etc.



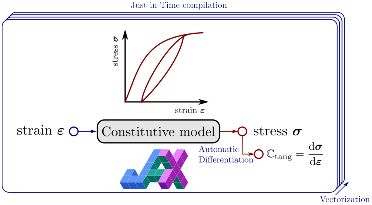
# A Python elasto-plastic behaviour

`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
```

## Outline

# 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

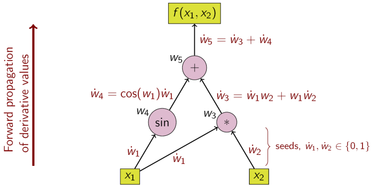
## Linear viscoelasticity

see also this COMET demo

```python
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**: $f'(x) = \dfrac{f(x+h) - f(x)}{h}$ with e.g. $h = 10^{-6}$
  **truncation/rounding errors,** $O(dim)$ **evauations**
- **Symbolic differentiatation**: $f$ 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]

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

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

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

## Differentiating through elastoplasticity

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

### Return mapping algorithm

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

- 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 $\Delta\boldsymbol{\varepsilon}^{\text{p}} = \Delta p \dfrac{3}{2\sigma_{\text{eq}}^{\text{elas}}}\boldsymbol{s}_{\text{elas}}$

$$\textbf{Solve } r(\Delta p) = \sigma_{\text{eq}}^{\text{elas}} - 3\mu\Delta p - R(p_n + \Delta p) = 0 \tag{2}$$

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

**Every step is differentiable** with AD, **except (2)**.

## Differentiating through elastoplasticity

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

### Return mapping algorithm

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

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

- 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 $\Delta\boldsymbol{\varepsilon}^{\text{p}} = \Delta p \dfrac{3}{2\sigma_{\text{eq}}^{\text{elas}}}\boldsymbol{s}_{\text{elas}}$

$$\textbf{Solve } r(\Delta p) = \sigma_{\text{eq}}^{\text{elas}} - 3\mu\Delta p - R(p_n + \Delta p) = 0 \tag{2}$$

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

**Every step is differentiable** with AD, **except (2)**.

### Algorithm unrolling

Any algorithm used to solve (2) can be written in JAX using loops, conditionals, etc. We can **differentiate through the algorithm** (*unrolling the algorithm iterations*).

# Outline

## Implicit automatic differentiation [Blondel et al., 2022]

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

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

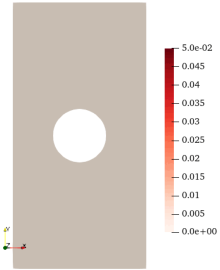
need only to solve a **linear system** for the **jacobian matrix** $[\partial_x F]$
the derivative computation becomes **independent from the algorithm** used to solve the nonlinear system, can use AD to form the jacobian $[\partial_x F]$
**Implementation of** `JAXNewton`

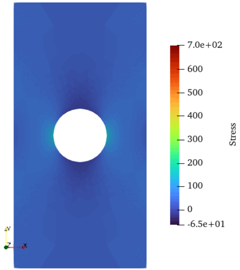
```
class JAXNewton:
    """A tiny Newton solver implemented in JAX.
    Derivatives are computed via custom implicit differentiation."""

    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)
```

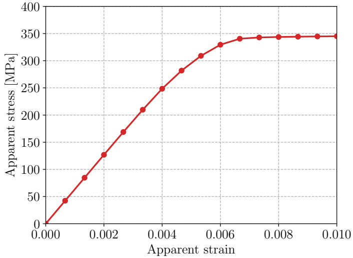
# Small-strain elastoplasticity

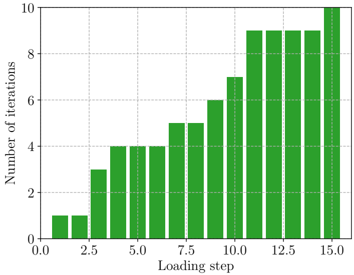
```python
@tangent_AD
def constitutive_update(self, eps, state, dt):
    deps = eps - state["Strain"]
    p_old = state["p"]
    mu = self.elastic_model.mu
    sig_el = state["Stress"] + self.elastic_model.C @ deps
    sig_eq_el = jnp.clip(self.equivalent_stress(sig_el), a_min=1e-8)
    n_el = dev(sig_el) / 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)
```

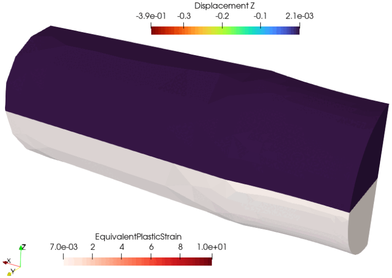
# $\boldsymbol{F}^{\mathrm{e}}\boldsymbol{F}^{\mathrm{p}}$ finite-strain plasticity

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

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

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

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



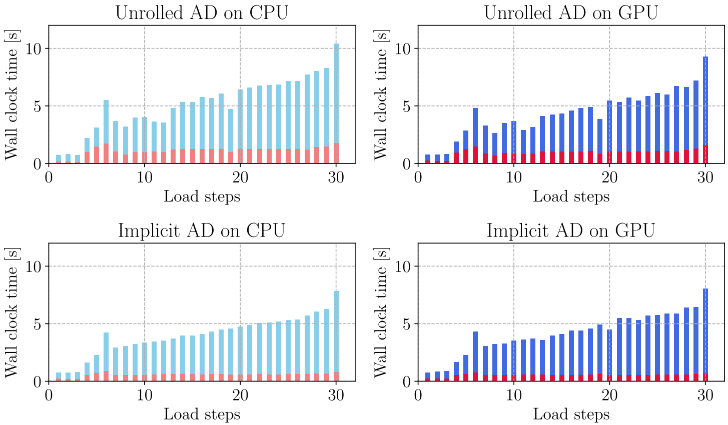

Resolution involves local solving of a Newton system of size 7
Tangent operator in $PK1/F$ using **implicit AD**:

$$\boldsymbol{P} = \boldsymbol{\tau}\boldsymbol{F}^{-1}$$

$$\mathbb{C}_{\mathrm{tang}} = \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}}$$

## $F^{e}F^{p}$ 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 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

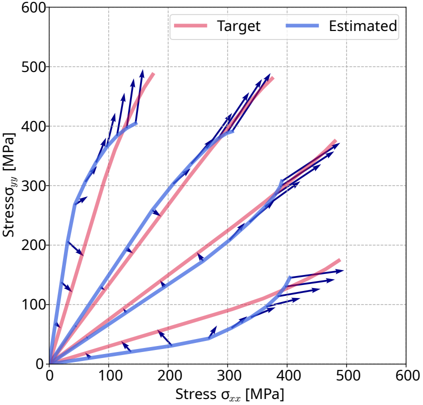
## Calibration

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

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

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

easy to obtain with **JAX**



## Integration within FEniCSx using External Operators

General concept of **non-UFL black-box** object implemented using **External Operator** [[Bouziani et al., 2021]]

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

$\boldsymbol{\sigma}(\nabla^s \boldsymbol{u})$ id defined through a UFL object `FEMExternalOperator`. The concrete behavior of the external operator is determined by a user program:

```
def sigma_call(epsilon: np.ndarray) -> np.ndarray:
    ...
    "<numerical algorithm>"
    ...
    return sigma_ # global vector of values at quadrature points
```

Inside of `sigma_call`, any external software can be used.

> *Andrey Latyshev, Jérémy Bleyer, Corrado Maurini, Jack S Hale. Expressing general constitutive models in FEniCSx using external operators and algorithmic automatic differentiation. 2024. https://hal.science/hal-04735022*

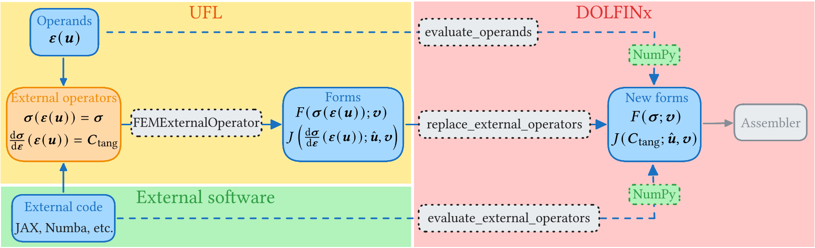
# Automatic differentiation of external operators

Forms containing external operators can be differentiated via `ufl.derivative`

```
J = ufl.derivative(F, u, ufl.TrialFunction(V))
J_expanded = ufl.algorithms.expand_derivatives(J)
```

The **derivative** of the external operator is a **new external operator**. The user must provide its concrete implementation:

```
def dsigma_call(epsilon: np.ndarray) -> np.ndarray:
    ...
    "<numerical algorithm>"
    ...
    return dsigma_ # global vector of values at quadrature points
```



# Conclusions and Outlook

`dolfinx_materials` **project** available at

https://github.com/bleyerj/dolfinx_materials

- **AD**: modern ML frameworks to **rethink material behavior libraries**
- **tangent operators** and material parameters **sensitivities**
- **implicit AD** is key

**External Operators** available at

https://github.com/a-latyshev/dolfinx-external-operator

## Outlook

- benchmarks on large-scale systems
- Neural network material models

## Conclusions and Outlook

`dolfinx_materials` **project** available at

https://github.com/bleyerj/dolfinx_materials

- **AD**: modern ML frameworks to **rethink material behavior libraries**
- **tangent operators** and material parameters **sensitivities**
- **implicit AD** is key

**External Operators** available at

https://github.com/a-latyshev/dolfinx-external-operator

**Outlook**

- benchmarks on large-scale systems
- Neural network material models

**Thank you for your attention!**