Non-linear local solver

Michal Habera, Andreas Zilian
Department of Engineering | University of Luxembourg

23rd August 2022 | FEniCS'22 conference

UNIVERSITÉ DU LUXEMBOURG

Contents

UNIVERSITÉ DU LUXEMBOURG

Motivation

Consider Laplace eqn. with non-linear conductivity law,

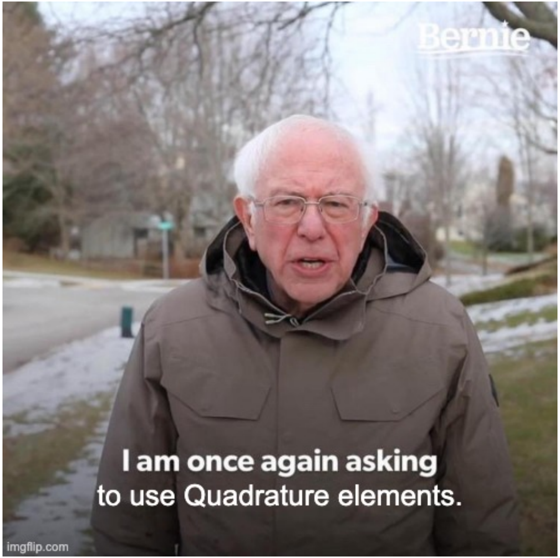
$$\nabla \cdot (\sigma(\varphi)\nabla\varphi) = 0, \tag{1}$$

$$f(\sigma, \varphi) = 0 \tag{2}$$

where $f(\sigma, \varphi)$ is a known implicit relation.

```
φ = Function(mesh, P1Space)
δφ = TestFunction(mesh, P1Space)
σ = Function(mesh, QuadratureSpace)
δσ = TestFunction(mesh, QuadratureSpace)

F_φ = ufl.inner(σ * ufl.grad(φ), ufl.grad(δφ)) * ufl.dx  # Eqn. (1)
F_σ = ufl.inner(f(σ, φ), δσ) * ufl.dx                     # Eqn. (2)
```


Bernie
I am once again asking
to use Quadrature elements.
imgflip.com

Motivation

Linearised problem:

$$\begin{bmatrix} \frac{\partial F_\varphi}{\partial \varphi} & \frac{\partial F_\varphi}{\partial \sigma} \\ \frac{\partial F_\sigma}{\partial \varphi} & \frac{\partial F_\sigma}{\partial \sigma} \end{bmatrix} \begin{bmatrix} \Delta\varphi \\ \Delta\sigma \end{bmatrix} = - \begin{bmatrix} F_\varphi \\ F_\sigma \end{bmatrix}$$

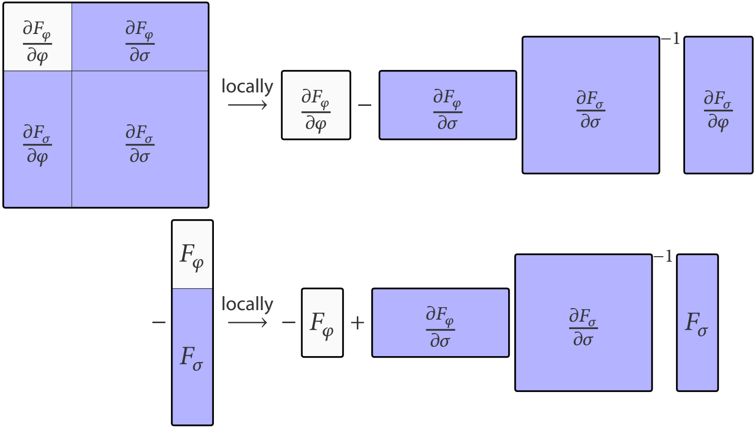
Cons:

- global problem size $N = N_\varphi + N_\sigma$,
- size of the augmented block N_σ depends on quadrature rule,
- there is no continuity to σ, so has even more DOFs globally,
- you almost certainly won't have a solver which scales linearly wrt. the problem size,
- complicated (non-linear) block structure makes it very challenging to find a good preconditioner.

Option 1: Schur condensation

1. Linearise.
2. Algebraically eliminate.

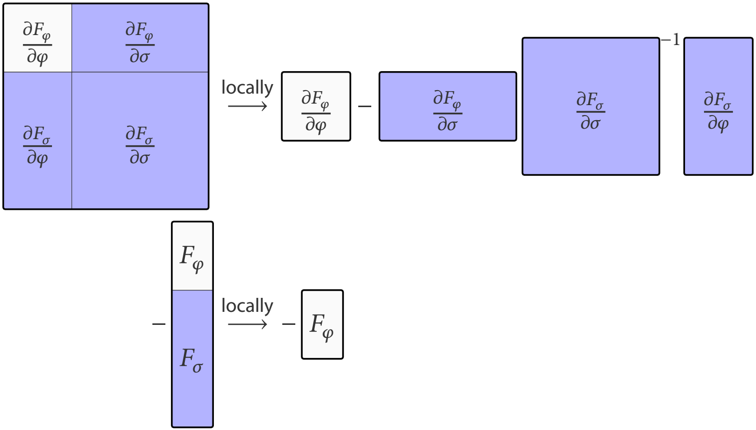
Note: Last iterate (φ_k, σ_k) satisfies both equilibria.



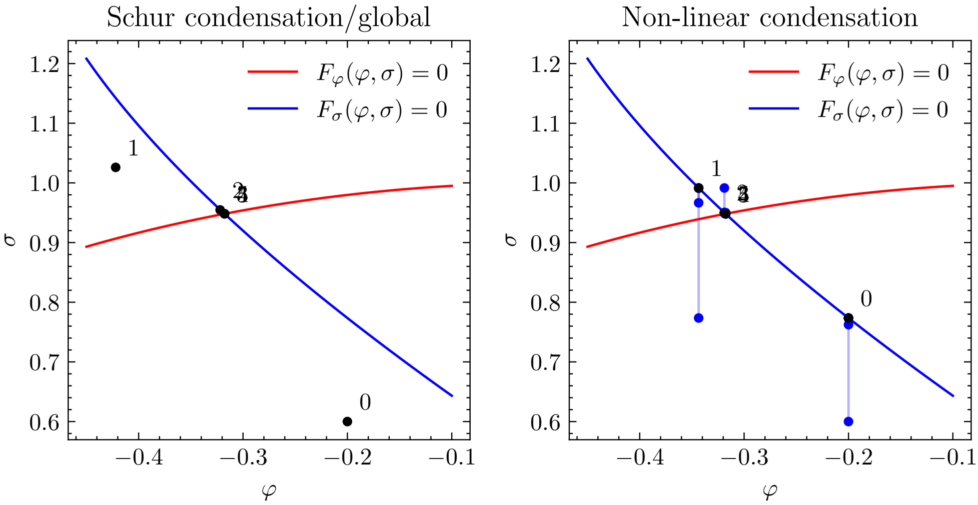
Option 2: Non-linear condensation

1. For a known global state φ_n find consistent local state σ_n s.t. $F_\sigma(\varphi_n, \sigma_n) = 0$.
2. Compute tangent consistent with this algorithmic dependence.

Note: Each iterate (φ_n, σ_n) satisfies local equilibrium.



Comparison


Schur condensation/global
Non-linear condensation
$F_\varphi(\varphi, \sigma) = 0$
$F_\sigma(\varphi, \sigma) = 0$
σ
φ
1.2
1.1
1.0
0.9
0.8
0.7
0.6
−0.4
−0.3
−0.2
−0.1
0
1
2
3
4


IS IT POSSIBLE IN FENICSX?
YOU NEED TO HOOK INTO FFCX GENERATED C KERNELS.
imgflip.com

Common interface

1. (locally) How to assemble condensed tangent?
2. (locally) How to assemble condensed residual?
3. (locally) How to reconstruct local state σ_n knowing global state φ_n?
4. (globally) How to update previous local state?

```
ls = dolfiny.localsolver.LocalSolver(
  function_space=[P1Space, QuadratureSpace],
  local_spaces_id=[1],
  J_integrals={...},
  F_integrals={...},
  local_integrals={...},
  local_update=...
)

problem = dolfiny.snesblockproblem.SNESBlockProblem(..., localsolver=ls)
```

Common interface

In essence, user wraps local kernel code and has information about all FFCx compiled kernels provided.

```
@numba.njit
  def J(A, J, F):
    # Adventurous user code here ...
    # A is NumPy array where you assign the result
    # J is list of lists of KernelData (tangents)
    # F is list of KernelData (residuals)
```

Provided information:

```
collections.namedtuple("KernelData", ("kernel",              # Callable kernel fn()
                                      "array",               # NumPy array where fn() assembles
                                      "w",                   # DOF values of all Coefficients
                                      "c",                   # Values of Constants
                                      "coords",              # Cell geometry
                                      "entity_local_index",
                                      "permutation",
                                      "constants",           # Information about Constants
                                      "coefficients",        # Information about Coefficients
                                      ))
```

Common interface

1 (locally) How to assemble condensed **tangent**?

```
@numba.njit
def J(A, J, F):
  A[:] = J[0][0].array - J[0][1].array @ np.linalg.solve(J[1][1].array,
                                                          J[1][0].array)
```

Internally provided for `SNESSetJacobian()`.

Common interface

2 (locally) How to assemble condensed **residual**?

---

for "Schur condensation":

```
@numba.njit
def F(A, J, F):
  A[:] = F[0].array - J[0][1].array @ np.linalg.solve(J[1][1].array,
                                                      F[1].array)
```

---

or "Non-linear condensation":

```
@numba.njit
def F(A, J, F):
  A[:] = F[0].array
```

Internally provided for `SNESSetFunction()`.

Common interface

3 (locally) How to reconstruct local state σ_n knowing global state φ_n?

---

for "Schur condensation":

```
@numba.njit
def solve_sigma(A, J, F):
  # Extract increment Δφ
  Δφ_idx = ...
  Δφ = F[1].w[Δφ_idx[0]:Δφ_idx[1]]

  # Extract local state σ
  σ_idx = ...
  σ = F[1].w[σ_idx[0]:σ_idx[1]]

  A[:] = σ - np.linalg.solve(J[0][0].array,
                             F[0].array - J[0][1].array @ Δφ)
```

Common interface

for ”Non-linear condensation”:

```
@numba.njit
def solve_sigma(A, J, F):
  # Extract local state σ
  σ_idx = ...
  σ = F[1].w[σ_idx[0]:σ_idx[1]]

  maxiter = 15
  for it in range(maxiter):
    F[1].array[:] = 0.0  # Re-evaluate residual
    F[1].kernel(F[1].array, F[1].w, F[1].c, F[1].coords,
                F[1].entity_local_index, F[1].permutation)
    if np.linalg.norm(F[1].array) < 1e-12:  # Check convergence
      break

    J[1][1].array[:] = 0.0  # Re-evaluate tangent
    J[1][1].kernel(J[1][1].array, J[1][1].w, J[1][1].c, J[1][1].coords,
                   J[1][1].entity_local_index, J[1][1].permutation)

    Δσ = np.linalg.solve(J[1][1].array, -F[1].array)
    σ += Δσ

    J[1][1].w[σ_idx[0]:σ_idx[1]] = σ  # Update local state for tangent
  A[:] = σ  # Copy over the final result
```

”Simple things should be simple, complex things should be possible.”

Alan Kay

Plasticity

Find displacement u, plastic strain increment $\Delta\varepsilon_p$ and plastic multiplier $\Delta\lambda$ s.t.

$$\nabla \cdot \sigma = 0, \quad \text{momentum balance,} \tag{3}$$

$$\Delta\varepsilon_p - \Delta\lambda \frac{\partial f}{\partial \sigma} = 0, \quad \text{flow rule,} \tag{4}$$

$$\min(\Delta\lambda, -f(\sigma)) = 0, \quad \text{equiv. to KKT conditions,} \tag{5}$$

where $\sigma = \sigma(u, \varepsilon_p)$ is stress tensor and $f = f(\sigma)$ is a yield function.

Plasticity

E.g. in 2D, $u \in P_2$ and 16-point quadrature rule the problem sizes are locally **(12, 48, 16)**,

$$
\begin{bmatrix}
\frac{\partial F_u}{\partial u} & \frac{\partial F_u}{\partial \varepsilon_p} & \frac{\partial F_u}{\partial \Delta\lambda} \\
\frac{\partial F_{\varepsilon_p}}{\partial u} & \frac{\partial F_{\varepsilon_p}}{\partial \varepsilon_p} & \frac{\partial F_{\varepsilon_p}}{\partial \Delta\lambda} \\
\frac{\partial F_{\Delta\lambda}}{\partial u} & \frac{\partial F_{\Delta\lambda}}{\partial \varepsilon_p} & \frac{\partial F_{\Delta\lambda}}{\partial \Delta\lambda}
\end{bmatrix}
$$

$$f_{\text{von mises}} = \sqrt{\frac{3}{2}\text{dev}(\sigma) : \text{dev}(\sigma)} - \sigma_y \qquad f_{\text{rankine}} = \max_i \sigma_i - \sigma_y \tag{6}$$

Local solver: 13k DOFs, monolithic: 205k DOFs.

Other applications

- static condensation/hybridisation,
- custom material laws, stress-strain relation provided as black-box, or with neural network (which is essentially a black-box),
- custom local solvers/return mappings for constrained optimisation - not limited to Newton method to solve local equilibrium,
- good ol' `LocalSolver` from legacy FEniCS,
- debugging and printing UFL operators.

Summary and outlook

Honest column:

- performance of internal Numba wrappers is terrible and has many limitations ⟶ needs a rewrite in C,
- condensation happens within the cell entity ⟶ across entities (facet-to-cell) or over patches is also interesting.