Nonlinear Energy Minimization with FEM and Automatic Differentiation

Michal Béreš

IG CAS and VSB-TUO

4 March 2026

Outline

Section 1: Problem Setup and FEM Structure

Problem Class and Discrete Objective

$$u = \arg\min_{v \in V} J(v),$$

$$u_h = \arg\min_{v \in V_h} J(v), \qquad V_h \subset V \text{ from an FEM discretization.}$$

- We solve large-scale nonlinear minimization problems from variational PDE models.
- Unknowns are free nodal degrees of freedom after Dirichlet elimination.
- Second-order methods require $g(u) = \nabla J(u)$ and $H(u) = \nabla^2 J(u)$ in finite-dimensional coordinates.
- Main challenge: keep exact second-order information while scaling to large sparse systems across FEM variants (P1/P2/Qk) and similar locality-driven, sparse energies.

Two Derivative Construction Paths

FEniCS/UFL path

- UFL = **Unified Form Language**.
- Write energy/weak form symbolically in UFL.
- Derive first/second variations symbolically from the UFL form.
- Assemble gradient and Hessian-related operators from those forms.

$$J \xrightarrow[\text{symbolic in UFL}]{} \nabla J,\ \nabla^2 J$$

AD code path (JAX/CasADi)

- Write $J(u)$ as executable array code.
- Use AD transformations for derivatives.
- Compute $g = \nabla J$ and matrix-free $H(u)p$ directly.
- Reconstruct sparse H from HVPs (Hessian-vector products) via coloring when needed.

$$J(u) \xrightarrow[\text{AD trace}]{} g(u),\ H(u)p$$

When Each Path Is Better

UFL-based approach is often best when

- the PDE model is naturally written as local variational forms,
- constitutive terms, sources, and boundary terms are all representable in UFL operators,
- your workflow is assembled-form first (standard Newton/PETSc pipeline).

AD + HVP (Hessian-vector products) + coloring is often best when

- constitutive response is algorithmic (return mapping, history variables, branching),
- energy evaluation includes nested/nonlocal loops (FE^2 = two-scale Finite Element Method, FFT = Fast Fourier Transform-based solvers, neighbor/bond interactions),
- learned/black-box blocks are present and you still need scalable matrix-free $H(u)p$ and sparse Hessian recovery.

Model Energies Used for Validation

- **p-Laplacian (2D):**

$$J(v) = \frac{1}{p} \int_\Omega \|\nabla v\|^p \, dx - \int_\Omega f v \, dx$$

  Convex nonlinear diffusion; primary scaling benchmark.

- **Ginzburg-Landau (2D):**

$$J(v) = \int_\Omega \left(\frac{\varepsilon}{2} \|\nabla v\|^2 + \frac{1}{4}(v^2 - 1)^2 \right) dx$$

  Smooth but stiff nonlinearity; stresses globalization choices.

- **Neo-Hookean hyperelasticity (3D):**

$$J(v) = \int_\Omega W(\nabla v) \, dx, \quad W(F) = C_1(I_1 - 3 - 2 \log \det F) + D_1(\det F - 1)^2$$

  Vector-valued and strongly nonlinear; most demanding robustness test.

Concrete Case Where AD-Code Path Wins

Example: FE^2 (two-scale FEM) with history-dependent micro response.

- At each macro quadrature point q, stress is produced by a *nested algorithm*:

$$(\sigma_q, D_q) = \texttt{MicroSolve}(\varepsilon_q, z_q^-),$$

  where `MicroSolve` may run local Newton iterations, branching, and state updates.
- This is not representable as one *pure UFL symbolic form*; it needs external/custom operators and manual derivative plumbing.
- UFL targets declarative weak forms/operators, not solver-in-the-integrand programs.
- In such FE^2 setups, one can use an **RNN surrogate** for the micro response.
- For AD-based pipelines, this is natural: the RNN block is just part of executable code, so derivatives (including Hessian-vector products) are obtained directly by AD.

Ref: Ghavamian & Simone, *CMAME* 357 (2019), 112594, doi:10.1016/j.cma.2019.112594.

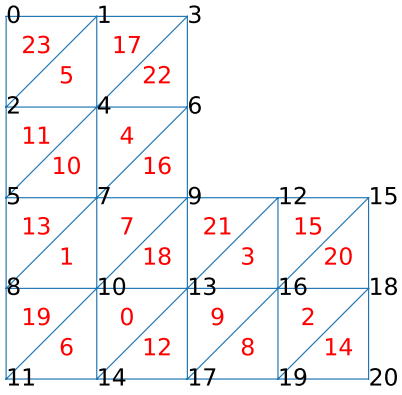
FEM Implementation Contract (General; P1 Example)

A typical local FEM energy kernel receives (P1 shown for concreteness):

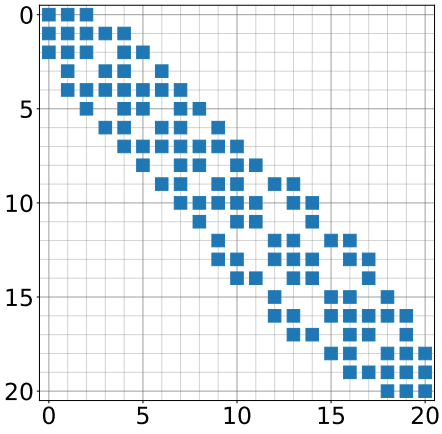
- `u`: free unknowns
- `u_0`: vector with Dirichlet values (zero elsewhere)
- `freedofs`: indices of free DoFs
- `elems`: element-to-node connectivity
- local basis/evaluation data on each element (for P1: `dvx,dvy` (and `dvz` in 3D))
- `vol`: element area/volume weights

Key structural fact: for local FEM discretizations, Hessian sparsity is mostly mesh/topology-driven and reusable.

Sparsity Comes from Mesh Locality



FEM mesh (example)



Hessian sparsity pattern

Nonzeros reflect overlapping basis supports.

Implementation Tracks

Track	Derivative Layer	Optimization / Linear Solves
MATLAB baseline	FD / analytic ∇J; SFD Hessian info from `fminunc`	Trust-region with diagonal-PCG
MATLAB + CasADi	Exact AD ∇J + exact HVP (`jtimes`)	TrustLS / TrustTR / Hybrid; AGMG (Notay) + Krylov/direct
Python + JAX	Exact AD ∇J + exact HVP (`grad`+`jvp`); `jit`	Same pipeline; SciPy sparse solvers + PyAMG

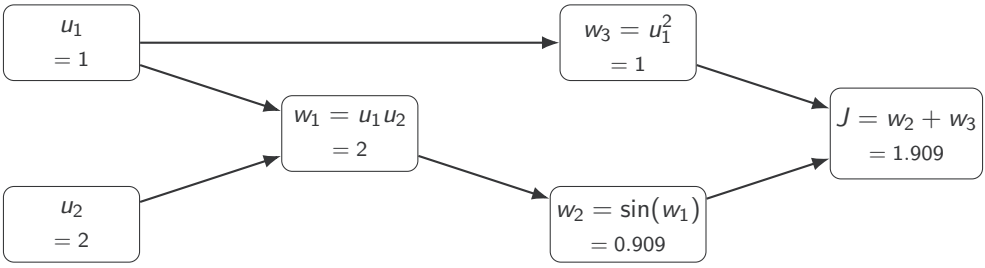
Section 2: Automatic Differentiation and Hessian-Vector Products

AD Operator Map: JAX and CasADi

Task	JAX call	CasADi call	Math object
Energy value	`J(u)` / `jit(J)(u)`	`Function('J',...)(u)`	$J(u)$
Value + grad	`value_and_grad(J)(u)`	`Function('Jg',...)(u)`	$(J, \nabla J)$
Gradient	`grad(J)(u)`	`gradJ=gradient(Jexpr,u)`	$\nabla J(u)$
JVP	`jvp(f,(u,),(v,))[1]`	`jtimes(f,u,v,false)`	$Df(u)\,v$
VJP	`vjp(f,u)` pullback	`jtimes(f,u,w,true)`	$Df(u)^\top w$
VJP	`vjp(f,u)[1](w)`	`jtimes(f,u,w,true)`	$Df(u)^\top w$
Hessian-vector	`jvp(grad(J),(u,),(p,))[1]`	`jtimes(gradJ,u,p,false)`	$\nabla^2 J(u)\,p$
Full Hessian	`jacfwd(jacrev(J))(u)`	`hessian(Jexpr,u)`	$\nabla^2 J(u)$

AD in one picture: compute $J(u)$ by a primal sweep

$$J(u_1, u_2) = \sin(u_1 u_2) + u_1^2 \qquad \text{(example: } u_1 = 1,\ u_2 = 2\text{)}$$



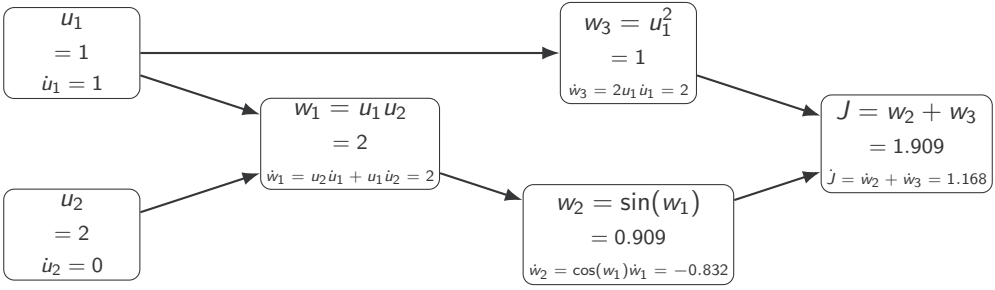
Primal (forward) evaluation: compute all intermediates once, then J.

Forward mode (JVP): primal sweep + tangent propagation

Dot notation means directional derivative along the seed direction p:

$$\dot{w}_i := Dw_i(u)[p], \qquad w_i = \phi_i(\{w_j\}_{j\in \mathrm{pred}(i)}) \;\Rightarrow\; \dot{w}_i = \sum_{j\in \mathrm{pred}(i)} \frac{\partial \phi_i}{\partial w_j}\dot{w}_j.$$

$$\dot{u}_1 = p_1, \quad \dot{u}_2 = p_2, \qquad \dot{J} = DJ(u)\,p \quad (\text{example: } u = (1,2),\ p = (1,0))$$



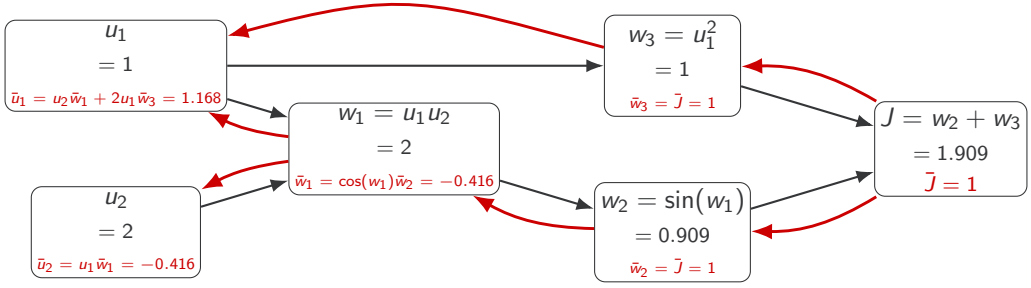
Interpretation: $\dot{J}$ is the directional derivative along p (one forward sweep $\Rightarrow$ one JVP).

Reverse mode (VJP): one backward sweep gives full ∇J

Adjoint pass: initialize all adjoints to zero ($\bar{u} = \bar{w} = 0$), and set $\bar{J} = 1$. Backward accumulation (children $\rightarrow$ parents):

$$\bar{w}_j \mathrel{+}= \sum_{i \in \mathrm{succ}(j)} \frac{\partial w_i}{\partial w_j} \bar{w}_i$$

$$\bar{u} = \nabla J(u) \quad \text{(example: } u = (1, 2) \Rightarrow \nabla J = (1.168,\ -0.416)\text{)}$$



Solid black arrows = primal graph, solid red arrows = reverse adjoint sweep.

Takeaway for large-scale FEM: AD modes and $H(u)p$

Assume $u \in \mathbb{R}^n$, $n \gg 1$, and scalar energy $J(u)$.

- **Forward mode (JVP):** one sweep yields $\dot{J} = DJ(u)\,p$ for a chosen direction p.
- **Reverse mode (VJP):** for scalar J, one backward sweep yields the full gradient $\nabla J(u)$.
- **Hessian-vector product:** avoid forming H explicitly; use forward-over-reverse.

For the full FEM program $J : \mathbb{R}^n \to \mathbb{R}$:

$$g(u) = \nabla J(u), \qquad H(u)p = Dg(u)[p] = \frac{d}{d\varepsilon} \nabla J(u + \varepsilon p)\bigg|_{\varepsilon=0}.$$

1. Evaluate $J(u)$ with the primal FEM program (element kernels + assembly).
2. Run reverse mode to obtain $g(u) = \nabla J(u)$.
3. Apply directional AD to g in direction p to get $H(u)p$, without materializing H.

Implementations: JAX uses `jvp(grad(J),...)`; CasADi uses `jtimes(gradient(...),...)`.

Regularity and Modeling Limits for J

- Classical Hessian interpretation requires $J \in C^2$ near the current iterate.
- Nonsmooth constructs ($|\cdot|$, max, contact switches, clipping) create undefined/discontinuous Hessians at these points; AD then gives branch-wise derivatives.
- AD removes finite-difference noise but does *not* remove conditioning issues of the underlying PDE/FEM model.
- JAX constraints: pure array programs with JAX primitives, shape-stable control flow for efficient `jit`.
- CasADi constraints: expression must be representable in SX/MX graph form; external black-box kernels need explicit callback/derivative support.
- Reverse-mode memory scales with taped operation graph size; checkpointing/rematerialization may be needed for very large problems.

Theoretical Cost Model: J, ∇J, and `HessMult`

Let C_J be one primal energy evaluation cost on a fixed mesh/topology (typically $C_J = \Theta(n_{\mathrm{el}})$ for local element/bond kernels).

$$C_{\nabla J} \approx C_J + C_R, \qquad C_{Hp} \approx C_J + C_R + C_F,$$

where C_R and C_F denote reverse/forward sweep costs on the gradient program.

Quantity	AD sweeps (conceptual)	Cost in C_J
$J(u)$	primal	$1 \cdot C_J$
$\nabla J(u)$	primal + reverse	$(1+\rho)C_J$
$H(u)p$	primal + reverse + forward	$(1+\rho+\phi)C_J$

- $\rho, \phi = \mathcal{O}(1)$ are empirical constants; shown multipliers are heuristics, not guarantees.
- JAX documentation describes JVP/VJP as small-constant-multiple FLOP overheads; wall time still depends on `jit`, dispatch, and memory traffic.

Direct full Hessian AD is much costlier at scale; matrix-free + coloring gives $m\, C_{Hp} + \mathcal{O}(\mathrm{nnz}(H))$, $m \ll n$. Sources: JAX docs (Advanced autodiff, Benchmarking), CasADi docs §3.9, Baur–Strassen (1983).

Section 3: Sparse Hessian Assembly by Graph Coloring

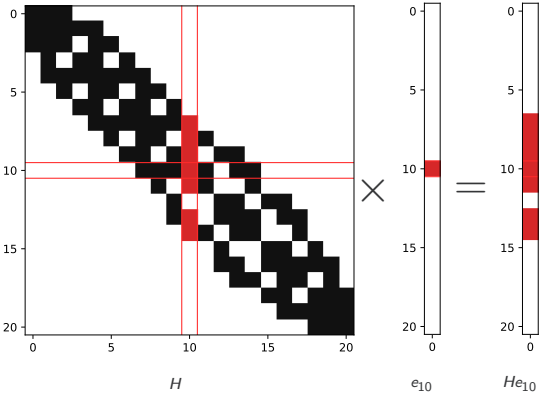
From HessMult to Sparse H

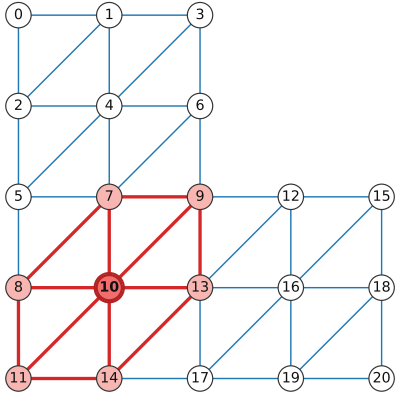
Assume access only to matrix-free products $v \mapsto Hv$. The goal is to recover an exact sparse Hessian using as few `HessMult` calls as possible.

- **One Probe in Matrix and Graph Views**
- **Overlap in $y = Hg$: why mixed rows appear**
- **Conflict Criterion from Overlap**
- **Conflict Graph and Coloring**
- **One Color Class as One Safe Probe**
- **Coloring Variants Used in the Code**
- **Practical Sparse Hessian Assembly Workflow**

1: One Probe in Matrix and Graph Views

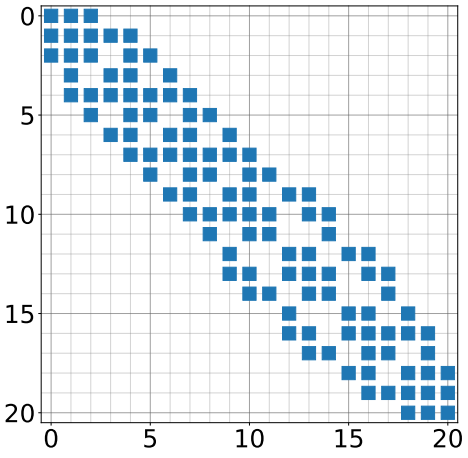
One probe $y = He_{10}$ reveals support of column 10 in H; the same support is node 10 plus one-hop neighbors in pattern($\mathcal{P}$).





pattern($\mathcal{P}$): node 10, one-hop neighbors, highlighted local couplings

2: Overlap in $y = Hg$: why mixed rows appear



Sparse pattern $\mathcal{P}$

- $\mathcal{P}_{ij} = 1 \iff H_{ij} \neq 0$; column j marks rows touched by variable u_j.
- In one grouped `HessMult`, $y = Hg$:

$$y_i = \sum_{j=1}^{n} H_{ij} g_j.$$

- If two active columns j, k satisfy $\mathcal{P}_{ij} = \mathcal{P}_{ik} = 1$, row i is mixed.
- Mixed rows are not separable from one probe, so safe groups must have disjoint row supports.

3: Conflict Criterion from Overlap

Define row-support sets $R_j = \{\, i : \mathcal{P}_{ij} = 1 \,\}$. Two columns conflict in one probe iff they overlap:

$$(j,k) \in E_{\text{conf}} \iff R_j \cap R_k \neq \emptyset \iff (\mathcal{P}^\top \mathcal{P})_{jk} > 0.$$

For symmetric Hessian patterns ($\mathcal{P}^\top = \mathcal{P}$):

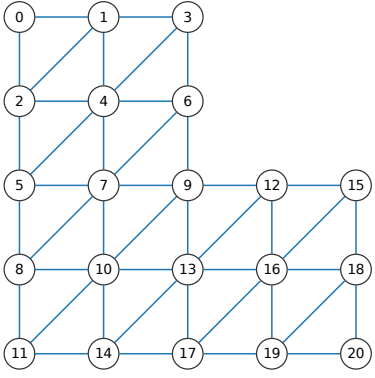
$$E_{\text{conf}} = \text{pattern}(\mathcal{P}^2) \setminus \text{diag}.$$

With the usual FE diagonal ($\mathcal{P}_{jj} = 1$), this gives the graph-distance rule:

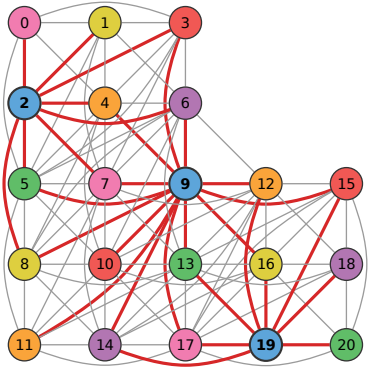
$$\text{dist}_{\mathcal{P}}(j,k) \leq 2 \implies j,k \text{ cannot share one probing vector.}$$

- Equivalent wording: vertices in $\mathcal{P}$ connected by a path of length 2 or shorter create overlap in `HessMult`.
- Coloring enforces this separation automatically.

4: Conflict Graph and Coloring

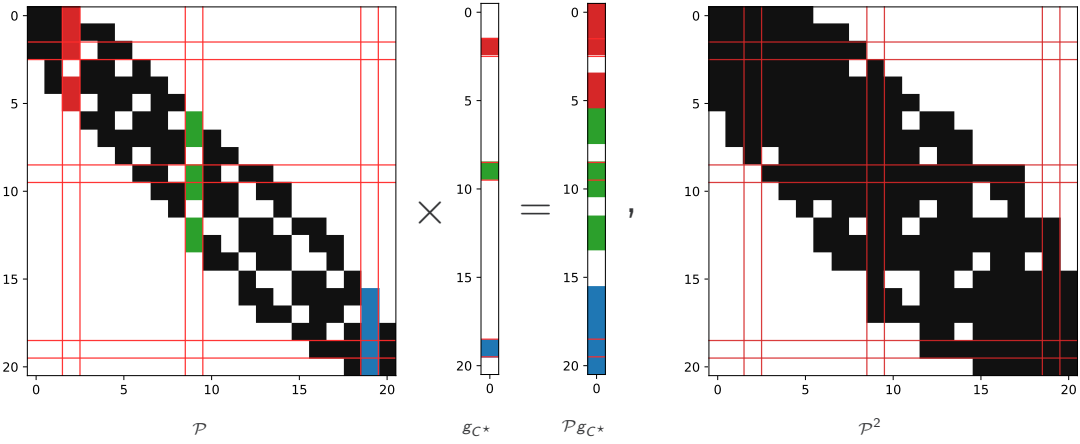


pattern($\mathcal{P}$): mesh-neighbor couplings



pattern($\mathcal{P}^2$): colored conflict graph; highlighted class used for one grouped probe

5: One Color Class as One Safe Probe



6: Coloring Variants Used in the Code

Input: adjacency $A \in \mathbb{R}^{n \times n}$
Build conflict structure $S = \text{pattern}(A^2)$ and neighbor lists $N(v)$
Initialize: $c(v) \leftarrow 1,\ s(v) \leftarrow 0,\ M \leftarrow \max_v |N(v)|$
Choose start vertex $v \leftarrow \arg\max_v |N(v)|$, set $s(v) \leftarrow 1,\ F \leftarrow N(v)$
for $t = 1, \ldots, n$ **do**
$v \leftarrow \left(\max_{u \in F} s_u > -\infty\right)\ ?\ \arg\max_{u \in F} s_u\ :\ \arg\max_{u \in V} s_u.$
$U \leftarrow \{c(u) : u \in N(v)\};\ c(v) \leftarrow \min\{k \in \{1, \ldots, M\} : k \notin U\}$
$s(N(v)) \leftarrow s(N(v)) + \mathbf{1};\ s(v) \leftarrow -\infty;\ F \leftarrow N(v)$
end for
Output: colors $c - 1$

Name	Main idea	Colors (2D/3D)	Time [s] (2D/3D)
MATLAB	Sequential first-fit: build each color class from non-overlapping columns.	14 / 87	253.86 / 87.76
Greedy	Greedy coloring on A^2: saturation-like vertex choice + first available color.	10 / 69	2.15 / 3.38
iGraph	Advanced coloring in optimized C backend.	7 / 68	2.74 / 17.26

7: Practical Sparse Hessian Assembly Workflow

Precompute once per mesh

1. Build pattern $\mathcal{P}$
2. Build graph of $\mathcal{P}^2$, then coloring
3. For each color c, build probe g_c
4. Build owner map $\pi_c(i)$: row $i \to$ unique column in c
5. Prebuild sparse index arrays (row, col) for fast writes

Each nonlinear iteration

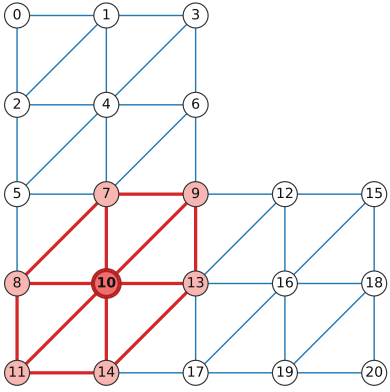
1. Evaluate $g = \nabla J(u)$
2. Compute $y_c = Hg_c$, $c = 1, \ldots, m$, via `HessMult`
3. For each stored pair (i, c): set $H_{i,\pi_c(i)} = (y_c)_i$
4. Enforce symmetry and finalize sparse format (CSR/CSC)

Result: exact sparse Hessian assembly with mesh-reusable setup and low per-iteration overhead.

Parallel Assembly Intuition: What Contributes to H_{ij}

$$J(u) = \sum_{e=1}^{N_e} J_e(u|_e), \quad g_i = \sum_{e \ni i} \frac{\partial J_e}{\partial u_i}, \quad H_{ij} = \sum_{e \ni i,\, j} \frac{\partial^2 J_e}{\partial u_i \partial u_j}.$$

- $J_e(u|_e)$ is the element energy contribution.
- $H_{ij} \neq 0$ only if DoFs i, j appear together in at least one element.
- For an owned row i, all required terms come from elements touching i (local overlap patch).
- After boundary-value exchange with neighbors (to obtain halo values), owned rows are assembled locally.



Example: node $i = 10$ and its one-hop neighborhood in pattern($\mathcal{P}$).
This is the local information needed to assemble row i of H.

Per-Rank Pipeline and Local Coloring (Small Graph)

For rank p:

1. Available u-data after exchange:

$$v_p = [u_{\text{owned},p}; u_{\text{halo},p}],$$

   where $u_{\text{halo},p}$ are neighbor values needed by local overlap elements.

2. Build a **small local conflict graph** only on

$$J_p = \text{owned}_p \cup \text{halo}_p, \qquad G_p = \text{pattern}\big(\mathcal{P}^2(J_p, J_p)\big)\,.$$

3. Color G_p locally $\Rightarrow$ local color classes $C_1, \ldots, C_{m_p}$.

4. For each local color c, use probe $t^{(c)} = \sum_{j \in C_c} e_j$, compute $r^{(c)} = \texttt{HessMult}(v_p, t^{(c)})$, and recover entries in owned rows.

5. Rank p assembles exactly entries H_{ij} with $i \in \text{owned}_p$ (all coupled j from local overlap).

Key point: coloring is **not global**; it is done on the local small graph G_p per rank.

Section 4: Optimization and Linear Solves

Globalization: Line Search, Trust Region, Hybrid

Line search

- Fast when direction is good
- Cheap control by $J(u + \alpha d)$
- Can fail with poor curvature

Trust region

- Robust under ill-conditioning
- Radius Δ_k controls model validity
- More expensive subproblems

Hybrid

- Use TR for safe direction
- Short line search for acceptance
- Best practical balance in hard 3D cases

Practical Hybrid Policy

1. Attempt Newton step from assembled sparse Hessian.
2. If curvature is poor or predicted decrease is unreliable, switch to trust-region subproblem.
3. If trust-region proposes a stable direction, run short line search on that direction.
4. Update trust radius from actual/predicted decrease ratio.
5. Stop based on gradient norm + model decrease, not step size alone.

This policy avoids false convergence and improves robustness in hyperelasticity.

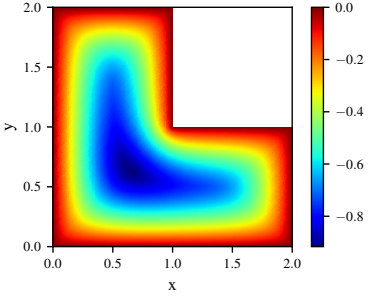
Each nonlinear iteration solves

$$H_k d_k = -g_k.$$

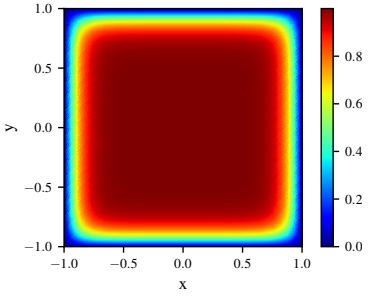
- Small/medium problems: sparse direct solver.
- Large problems: Krylov + preconditioning.
- Default robust choice for elliptic FEM-like operators: AMG-preconditioned Krylov.
- Reuse opportunities:
  - mesh-dependent hierarchy setup,
  - sparsity pattern and indexing maps,
  - compiled derivative kernels.

Section 5: Benchmarks and Results

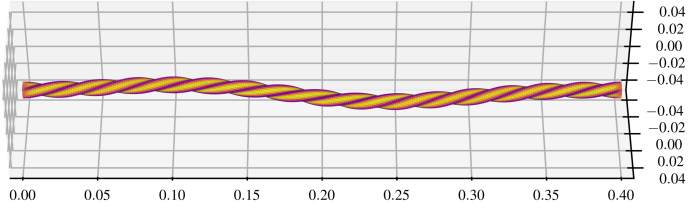
Benchmark Reference Solution Visualizations



p-Laplacian



GL



Hyperelasticity

Summary: original MATLAB (SFD) vs Python/JAX (AD)

Benchmark	DoFs	Python [s]	MATLAB [s]	Speedup	Iters
p-Laplace 2D	784,385	19.96 / 4.41	429.12	17.6	9 / 13
Ginzburg-Landau 2D	1,046,529	26.63 / 5.76	653.08	20.2	6 / 9
Hyperelasticity 3D ($t = 24$)	77,517	171.82	1837.23	10.7	49 / 128

For p-Laplacian and Ginzburg-Landau, Python time is reported as solve/setup. Speedup uses total Python time.

Cross-Framework Comparison

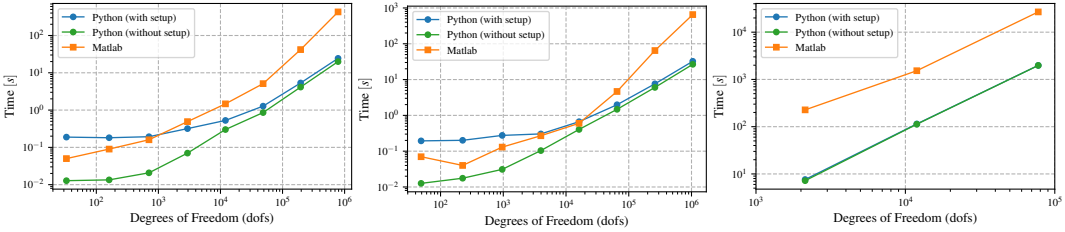
p-Laplace 2D, level 8 (784,385 DoFs)

Method	Time [s]	Iters	$J(u)$
Python/JAX (setup+solve)	24.37	9	-7.9600
New MATLAB (trustLS)	32.36	7	-7.9600
FEniCS serial	15.740	9	-7.9600
FEniCS parallel	3.925	9	-7.9600

Hyperelasticity 3D, level 3 (77,517 DoFs), step $t = 24$

Method	Time [s]	Iters	$J(u)$
Python/JAX	171.82	49	93.7039
MATLAB legacy	1837.23	128	93.7216
MATLAB approx/trustLS	4729.6	284	93.7041
MATLAB exact/trustLS	241.3	86	93.7039
MATLAB CasADi/trustLS	146.68	54	93.7039

Scaling Curves Across the Three Benchmarks



p-Laplacian | Ginzburg-Landau | Hyperelasticity

Python version shows near-linear scaling in DoFs, while the MATLAB version shows much worse scaling, largely due to the coloring algorithm and preconditioning.

Section 6: Conclusions

Implementation Recipe (Both Tracks)

1. Implement only $J(u)$ with vectorized FEM kernels.
2. Obtain exact g and `HessMult` via AD (CasADi or JAX).
3. Precompute mesh pattern, coloring, and extraction indices once.
4. Assemble sparse H only when matrix-based linear solves are useful.
5. Use line search / trust region / hybrid based on local robustness.
6. Match linear solver and preconditioner to problem size and conditioning.

Takeaways

- Core message: **AD + sparse Hessian assembly by graph coloring + robust optimization** is a practical path for large nonlinear FEM minimization.
- MATLAB+CasADi and Python+JAX can share the same algorithmic design.
- Benchmark evidence confirms large-scale gains while preserving solution quality.
- A reproducibility package and benchmark scripts are available in the public project repository. github.com/Beremi/nonlinear_energies_python