Adaptive FEM in Python — Error-Driven Mesh Refinement (2D Poisson)

Adaptive FEM in Python — Error-Driven Mesh Refinement (2D Poisson)
Author: Sourabh More
Date: 2025-09-12

Abstract

I built a small, end-to-end Adaptive Finite Element Method (FEM) solver for the 2D Poisson equation using error-driven mesh refinement (AMR). The pipeline is:
mesh → assemble sparse K → solve Ku=b → a-posteriori error (ZZ) → bulk mark → longest-edge bisection → repeat.
Even with a compact Python implementation (NumPy/SciPy/Matplotlib), the method reliably concentrates elements where the solution bends most and drives the estimated error down with each cycle. This report documents the problem, method, implementation choices, and results, with practical notes on what worked and what I'd improve next.

1. Problem

We solve on the unit square $\Omega=(0,1)\times(0,1)$:

$$-\Delta u = f(x, y) \text{ in } \Omega, \quad u = 0 \text{ on } \partial\Omega$$

with a smooth forcing: $f(x, y) = 10\cdot\sin(\pi x)\cdot\sin(\pi y)$.
Interpretation: think of u as a temperature field clamped to zero on the boundary, heated internally by f. (For intuition: the exact solution for this f is $u(x,y) = (5/\pi^2)\cdot\sin(\pi x)\cdot\sin(\pi y)$. I didn't need it for the algorithm, but it's useful for sanity checks.)

2. Method

1) Discretize — Start from a uniform grid on $(0,1)^2$. Split each rectangle into two triangles; use P1 (linear) basis functions.
2) Assemble — For each triangle T: $K_T = |T|\cdot(B^TB)$, where columns of B are the constant gradients of the shape functions on T. Load via centroid rule. Scatter-add into a sparse CSR matrix K and vector b.
3) Apply Dirichlet BC — Overwrite rows for boundary nodes to enforce u=0.
4) Solve — Use scipy.spsolve to get nodal values u.
5) Estimate error (ZZ) — Compute element gradients ∇u_h, recover a smooth nodal gradient by averaging, and define $\eta_T^2 \approx |T|\cdot||\nabla u_h - \hat{g}||^2$.
6) Mark & refine — Dörfler bulk marking (pick smallest set with ~θ fraction of total error), then longest-edge bisection; refine neighbors sharing the bisected edge to keep the mesh conforming.
7) Repeat — Re-assemble, re-solve, re-estimate; stop after a few cycles or when error is small.

3. Method (details)

3.1 Mesh and connectivity

- Initial grid: n_x × n_y rectangles, each split by the same diagonal into two triangles.
- Node ordering is row-major. Triangle connectivity is stored as integer triplets [i, j, k].
- This keeps memory light and plays well with sparse assembly.

3.2 Element matrices and load

- Triangle area: $|T| = 0.5 * |(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)|$.
- B ∈ R^{2×3} collects constant gradients of P1 shape functions.
- Element stiffness: $K_T = |T| \cdot (B^T B)$.
- Element load (centroid rule): $b_T \approx f(x_c, y_c) \cdot |T| / 3 \cdot (1, 1, 1)^T$.

3.3 Boundary conditions

Dirichlet u=0 is applied by overwriting rows for boundary nodes: set diagonal to 1, RHS to 0 on those rows.

3.4 Linear solve

The global system Ku=b is symmetric positive definite. For these sizes, SciPy's sparse direct solver is fine. Switching to CG + preconditioning is an easy extension.

3.5 ZZ a-posteriori error estimator

- Compute element gradient $g_T = \nabla u_h|_T$.
- Compute recovered nodal gradient $\hat{g}_i$ by averaging g_T over incident elements.
- For element T=(i,j,k), take $\bar{g}_T = (\hat{g}_i + \hat{g}_j + \hat{g}_k) / 3$.
- Indicator: $\eta_T^2 \approx |T| \cdot || g_T - \bar{g}_T ||^2$.
- Global estimate: $\eta = \mathrm{sqrt}(\Sigma_T \eta_T^2)$.

3.6 Marking and refinement

- Bulk (Dörfler) marking: sort η_T descending and pick the smallest set whose sum reaches $\theta \cdot \Sigma_T \eta_T$ (default θ=0.5).
- Longest-edge bisection: split each marked triangle across its longest edge. To avoid hanging nodes, also refine neighbors that share the split edge.

Pseudo-code:

```
for it in 0..cycles:
   assemble K, b
   apply Dirichlet
   solve Ku = b
   estimate elem error {η_T} and global η
   if it == cycles: break
   marked = bulk_mark({η_T}, theta)
   mesh = refine_longest_edge(mesh, marked, ensure_conformity=True)
```

4. Implementation notes

- Language/Libraries: Python 3 + NumPy + SciPy (sparse) + Matplotlib.
- Data structures: triangles as int arrays; assembly into LIL, convert to CSR for solving.
- Plotting: tricontourf for fields; simple line plots for meshes.
- Colab footnote: Colab's working dir is /content; saving to relative paths avoids path issues.

5. Results

All runs used $f(x,y)=10\cdot\sin(\pi x)\cdot\sin(\pi y)$ and $u=0$ on the boundary.
Visuals:

- Mesh (cycle 0 → cycle N): starts uniform and selectively densifies where curvature is higher.
- Solution field u(x,y): max near center, 0 at edges.
- Estimated error heatmap: bright where refinement is requested.
- Convergence curve: estimated error vs DOFs (nodes) on log–log scale, trending down.

Typical console trace:

```
[Cycle 0] N=   225  Tris=   392  est_err=1.01e-01  solve_time=0.001s
[Cycle 1] N=   361  Tris=   640  est_err=6.80e-02  solve_time=0.002s
[Cycle 2] N=   561  Tris=  1004  est_err=4.61e-02  solve_time=0.003s
[Cycle 3] N=   865  Tris=  1568  est_err=3.18e-02  solve_time=0.005s
```

6. Ablations

- Starting mesh (nx0, ny0): finer start → nicer first plots, more work upfront.
- Cycles: with $\theta=0.5$, 3–5 cycles show the pattern clearly.
- θ (marking fraction): 0.3 very focused (more cycles), 0.7 broader (fewer cycles but more elements).

7. Limitations

- Estimator simplicity: ZZ is easy and effective; residual-based indicators may be tighter on some problems.
- Solver choice: direct is fine for small/medium; large problems need CG/AMG.
- Refinement strategy: longest-edge is robust but not anisotropic.

8. Future work

- Switch to CG + Jacobi/AMG; track iteration counts and wall time.
- Implement residual-based indicators; compare to ZZ.
- Add triangle-quality metrics and optional smoothing.
- Extend to anisotropic diffusion or image-processing use cases.
- Use the exact solution to report L2 and H1 errors alongside the estimator.

10. References

- Zienkiewicz & Zhu (1992): superconvergent patch recovery and a-posteriori error estimates.
- Verfürth (1996): a posteriori error estimation and AMR techniques.
- SciPy docs (sparse matrices & solvers); classic P1 FEM notes.