# Numerical Methods for the Solution of PDEs

## Laboratory with deal.II — www.dealii.org

### Manufactured solutions, global refinement, measuring error rates

**Luca Heltai <luca.heltai@unipi.it>**

https://luca-heltai.github.io/nmpde
https://github.com/luca-heltai/nmpde

# *How to measure the Error?*

- Method of Manufactured Solutions
  - Take the “u” you want as a solution, plug in the equations, get the boundary conditions and the right hand side that force the given “u”
  - Integrate (with a fine quadrature formula) the difference between the exact solution and the computed one (VectorTools::integrate_difference, or helper classes)
  - Possibly integrate the difference between the gradients of the exact and computed solutions

# *Error Estimates*

Local Estimate:

$$\|u - \Pi u\|_{s,T_m} \lesssim \rho_m^{-s} h_m^{k+1} \, | u |_{k+1,T_m}$$

Global Estimate (for quasi uniform triangulations):

$$\sum_m \left( \|u - \Pi u\|_{s,T_m} \right) \lesssim h^{k+1-s} \, | u |_{k+1,\Omega}$$

# *Error Estimates*

Local Estimate:

$$\|u - \Pi u\|_{s,T_m} \lesssim \rho_m^{-s} h_m^{k+1} \, | u |_{k+1,T_m}$$

If $V_h \subset H^s(\Omega)$ and Triangulation is *quasi-uniform*

$$\|u - \Pi u\|_{s,\Omega} \lesssim h^{k+1-s} \, | u |_{k+1,\Omega}$$

## *To Reduce the Error:*

- Globally, the error is dominated by *largest* element of the mesh and the $H^{k+1}(\Omega)$ norm of the exact solution

  - Reduce the overall size of the mesh $h$ (**global refinement**), when we don't know the $H^{k+1}(\Omega)$ norm of the exact solution

  - Reduce the size of the elements where the solution has large $H^{k+1}(\Omega)$ norm, or where we estimate that $H^{k+1}(\Omega)$ norm of the solution would be large (**local refinement**)

# *Estimate the rate of convergence*

- Once you have computed the error, how do we measure if we get the correct *convergence ratio?*

- *Consider Poisson Problem.* $V := H^1(\Omega)$

$$\| u - u_h \|_1 \lesssim \| u - \Pi u \|_1 \lesssim h^1 |u|_{2,\Omega}$$

$$\| u - \Pi u \|_0 \lesssim h^2 |u|_{2,\Omega}$$

**We still need to prove that we can use $u_h$ in the last estimate!**

# *Estimate the rate of convergence*

- Compute two successive solutions, on half the size of the mesh (i.e., after one global refinement):

$$\| u - u_{2h} \| \sim \tilde{C}(2h)^p$$

$$\| u - u_h \| \sim \tilde{C}(h)^p$$

$$\frac{\| u - u_{2h} \|}{\| u - u_h \|} \sim 2^p$$

$$p \sim \log_2\left(\frac{\| u - u_{2h} \|}{\| u - u_h \|}\right)$$

# *Back to C++*

- Today's program:
  - Poisson for general coefficients, boundary data, and rhs
  - Work on successively refined grids
  - Estimate $L^2(\Omega)$ and $H^1(\Omega)$ errors

# *Poisson problem revisited*

Homogeneous Dirichlet case, constant coefficient equal to 1:

$$-\Delta u = f \qquad \text{in } \Omega$$
$$u = 0 \qquad \text{on } \partial\Omega$$

$$\gamma_\Gamma : H^1(\Omega) \mapsto H^{\frac{1}{2}}(\Gamma)$$ Trace operator

$$V := H^1_0(\Omega) := \{v \,|\, v \in L^2(\Omega), \nabla v \in L^2(\Omega), \gamma_{\partial\Omega} v = 0\}$$

Weak form: given $f \in V^*$, find $u \in V$ such that

$$(\nabla u, \nabla v) = (f, v) \qquad \forall v \in V$$

# *Poisson problem revisited*

Non-homogeneous Dirichlet case, constant coefficient equal to 1:

$$
\begin{aligned}
-\Delta u &= f && \text{in } \Omega \\
u &= g && \text{on } \partial\Omega
\end{aligned}
$$

$$V_0 := H_0^1(\Omega) := \{v \,|\, v \in L^2(\Omega), \nabla v \in L^2(\Omega), \gamma_{\partial\Omega} v = 0\}$$

$$V_g := V_0 + u_D$$ Where $\gamma_{\partial\Omega} u_D = g$

Weak form: given $f \in V^*$, find $u \in V_g$ such that

$$(\nabla u, \nabla v) = (f, v) \qquad \forall v \in V_0$$

# *Poisson problem revisited*

Mixed boundary conditions, non-constant coefficients

$$
\begin{aligned}
-\nabla \cdot (a \nabla u) &= f && \text{in } \Omega \\
u &= g_D && \text{on } \Gamma_D \\
n \cdot (a \nabla u) &= g_N && \text{on } \Gamma_N
\end{aligned}
$$

$$V_{0,\Gamma_D} := H^1_0(\Omega) := \{v \,|\, v \in L^2(\Omega), \nabla v \in L^2(\Omega), \gamma_{\Gamma_D} v = 0\}$$

$$V_{g_D,\Gamma_D} := V_{0,\Gamma_D} + u_D$$ Where $\gamma_{\Gamma_D} u_D = g_D$

Weak form: given $f \in V^*_{0,\Gamma_D}$, find $u \in V_{g_D,\Gamma_D}$ such that

$$(a \nabla u, \nabla v) = (f, v) + \int_{\Gamma_N} g_N v \qquad \forall v \in V_{0,\Gamma_D}$$

# *Trial spaces VS test spaces*

$$V_{0,\Gamma_D} := H^1_0(\Omega) := \{v \,|\, v \in L^2(\Omega), \nabla v \in L^2(\Omega), \gamma_{\Gamma_D} v = 0\}$$

$$V_{g_D,\Gamma_D} := V_{0,\Gamma_D} + u_D$$ Where $\gamma_{\Gamma_D} u_D = g_D$

Weak form: given $f \in V^*_{0,\Gamma_D}$, find $u \in V_{g_D,\Gamma_D}$ such that

$$(a\nabla u, \nabla v) = (f, v) + \int_{\Gamma_N} g_N v \qquad \forall v \in V_{0,\Gamma_D}$$

CANNOT apply Lax-Milgram: $V_{0,\Gamma_D} \neq V_{g_D,\Gamma_D}$

# *Trial spaces VS test spaces*

$$V_{0,\Gamma_D} := H_0^1(\Omega) := \{v \,|\, v \in L^2(\Omega), \nabla v \in L^2(\Omega), \gamma_{\Gamma_D} v = 0\}$$

$$V_{g_D,\Gamma_D} := V_{0,\Gamma_D} + u_D$$

Where $\gamma_{\Gamma_D} u_D = g_D$

Weak form: given $f \in V^*_{0,\Gamma_D}$, find $u_0 \in V_{0,\Gamma_D}$ such that

$$(a\nabla u_0, \nabla v) = (f, v) + \int_{\Gamma_N} (g_N - n \cdot (a\nabla u_D))v - (a\nabla u_D, \nabla v) \qquad \forall v \in V_{0,\Gamma_D}$$

Write $u = u_0 + u_D$ (now we can apply Lax-Milgram)

$u_D$ is arbitrary, and such that $\gamma_{\Gamma_D} u_D = g_D$

# *How to implement* $V_{g_D,\Gamma_D}$, $V_{0,\Gamma_D}$?

- Option 1 (**not implemented in deal.II**):
  encode in DoFHandler (n_dofs of $H^1_{0,\Gamma_D}(\Omega)$ < n_dofs of $H^1(\Omega)$)
  and in basis functions (i.e., $\gamma_{\Gamma_D} v_i = 0 \quad \forall v_i \in V_h$)

- Option 2 (Penalty methods, Lagrange multipliers):
  impose boundary conditions weakly

- Option 3 (Algebraic approach: strong imposition):
  post-process Linear systems, solution vectors, and rhs vectors to **set to** $g_D$
  degrees of freedom with support points on $\Gamma_D$

# *Algebraic approach*

- Main idea: assemble matrix $\tilde{A}_{ij} := (a \nabla v_j, \nabla v_i)$

  and right-hand-side $\tilde{F}_i := (f, v_i) + \int_{\Gamma_N} g_N v_i$

- split dofs $u = \begin{pmatrix} u_{\Omega \cup \Gamma_N} \equiv u_O \\ u_C \end{pmatrix}$ $\tilde{F} = \begin{pmatrix} F_O \\ F_C \end{pmatrix}$

- and matrix $\tilde{A} = \begin{pmatrix} A_{OO} & A_{OC} \\ A_{CO} & A_{CC} \end{pmatrix}$

- where “C” stands for “constrained”

# *Mimic continuous approach*

- compute $g_D$, using VectorTools::interpolate_boundary_values

- eliminate row “C” from $\tilde{A}$, and set rhs $\tilde{F}_C \mapsto g_D$:
$$\begin{pmatrix} A_{OO} & A_{OC} \\ 0 & I_{CC} \end{pmatrix} \begin{pmatrix} u_O \\ u_D \end{pmatrix} = \begin{pmatrix} \tilde{F}_O \\ g_D \end{pmatrix}$$

- “move” $A_{OC}$ to rhs to restore symmetry in matrix:
$$\begin{pmatrix} A_{OO} & 0 \\ 0 & I_{CC} \end{pmatrix} \begin{pmatrix} u_O \\ u_D \end{pmatrix} = \begin{pmatrix} \tilde{F}_O - A_{OC} g_D \\ g_D \end{pmatrix}$$

- rescale $I_{CC}$ for conditioning:
$$\begin{pmatrix} A_{OO} & 0 \\ 0 & \alpha I_{CC} \end{pmatrix} \begin{pmatrix} u_O \\ u_D \end{pmatrix} = \begin{pmatrix} \tilde{F}_O - A_{OC} g_D \\ \alpha g_D \end{pmatrix}$$

MatrixTools::apply_boundary_values

$$\tilde{A} \mapsto \begin{pmatrix} A_{OO} & 0 \\ 0 & \alpha I_{CC} \end{pmatrix} \qquad u \mapsto \begin{pmatrix} u_O \\ u_D \end{pmatrix} \qquad \tilde{F} \mapsto \begin{pmatrix} \tilde{F}_O - A_{OC} g_D \\ \alpha g_D \end{pmatrix}$$

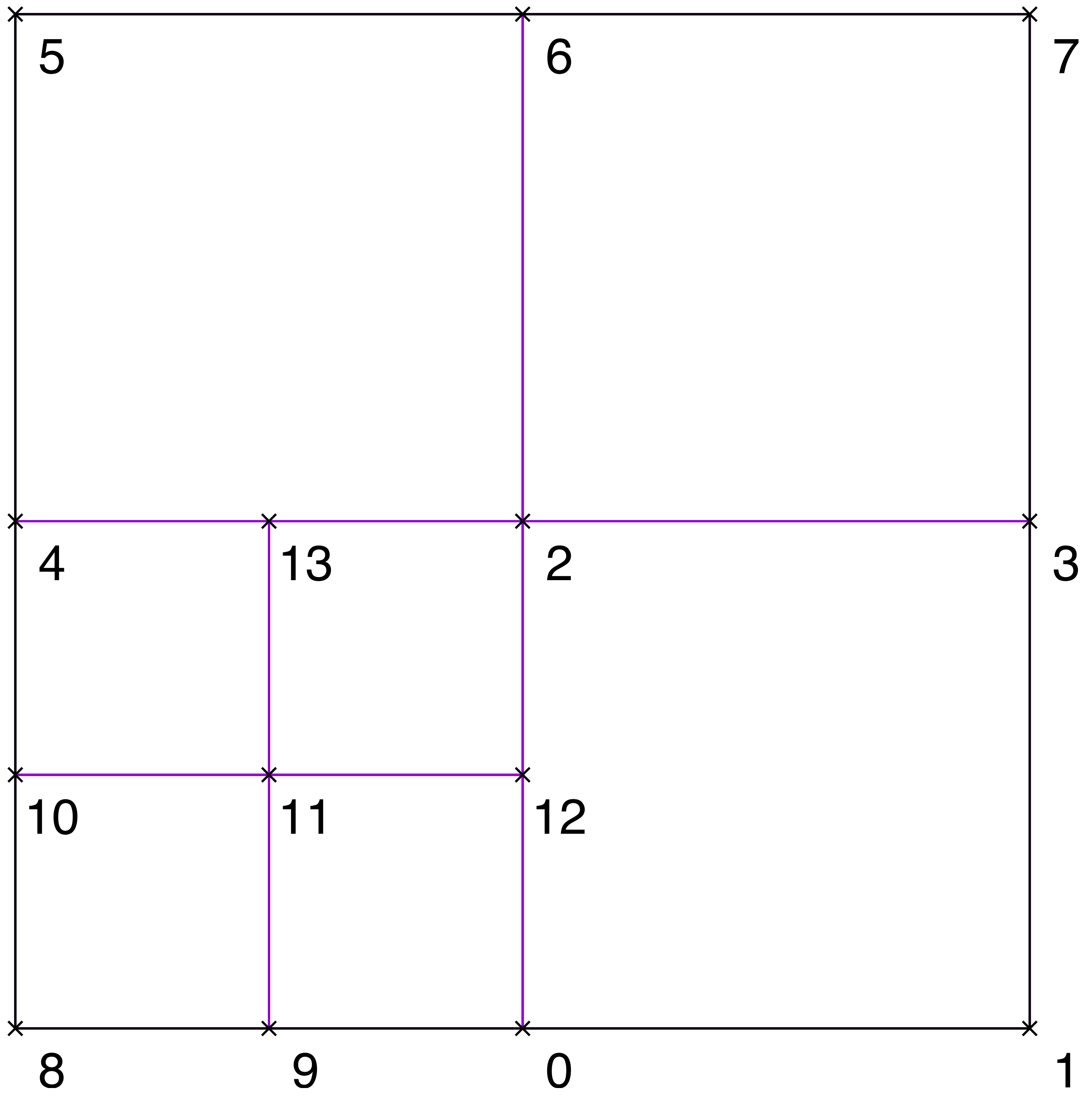
# *Special case of  AffineConstraints*

- General case: constrained dofs are a subset of all dofs $\mathcal{N}_C \subset \mathcal{N}$
- AffineConstraints: $x_i = \sum_{j \in \mathcal{N} \setminus \mathcal{N}_C} C_{ij} x_j + b_i \qquad \forall i \in \mathcal{N}_C$
- Algebraic solution can be performed efficiently as a three-step process:
    - Condense
    - Solve
    - Distribute (only needed if $C \neq 0$)

# Condense-Solve-Distribute

- Given, $\tilde{A} = \begin{pmatrix} A_{OO} & A_{OC} \\ A_{CO} & A_{CC} \end{pmatrix}$, $\tilde{F} = \begin{pmatrix} F_O \\ F_C \end{pmatrix}$, and constraints $u_C = Cu_O + b$
- Take constraints into accounts in “O”: $A_{OO}u_O + A_{OC}u_C = (A_{OO} + A_{OC}C)u_O + A_{OC}b = F_O$
- Ignore rows “C” in matrix and rhs and solve $Au = F$ where
  - $\tilde{A} = \begin{pmatrix} A_{OO} & A_{OC} \\ A_{CO} & A_{CC} \end{pmatrix} \mapsto A = \begin{pmatrix} A_{OO} + A_{OC}C & 0 \\ 0 & \alpha I_{CC} \end{pmatrix}$
  - $\tilde{F} = \begin{pmatrix} F_O \\ F_C \end{pmatrix} \mapsto F = \begin{pmatrix} F_O - A_{OC}b \\ \alpha b \end{pmatrix}$
- Distribute constraints: $u = \begin{pmatrix} u_O \\ b \end{pmatrix} \mapsto u = \begin{pmatrix} u_O \\ Cu_O + b \end{pmatrix}$

# *Hanging nodes*

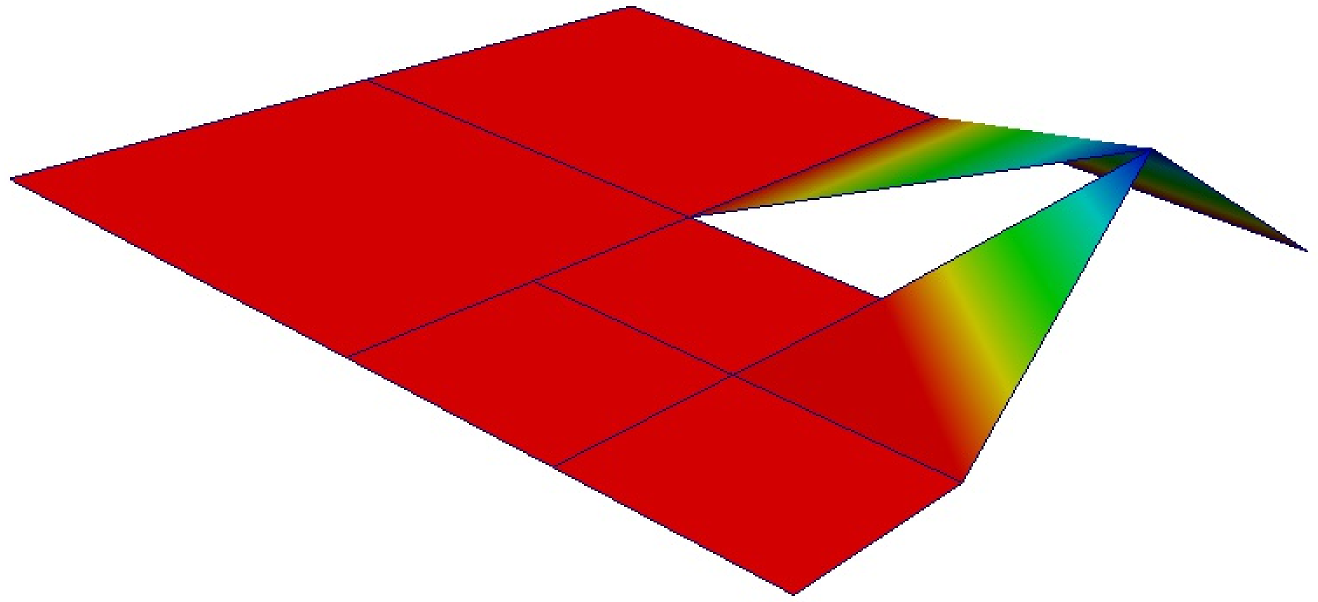



Discontinuous FE space!

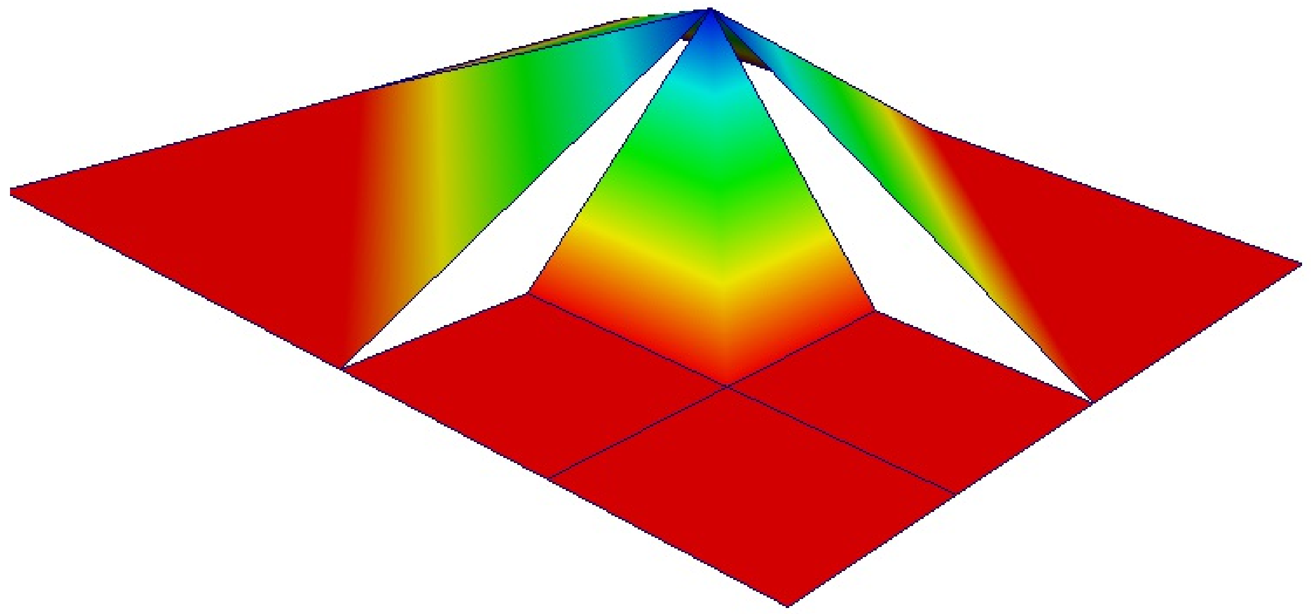
Not a subspace of $H^1$

Bilinear forms would require special treatment as gradients are not defined everywhere

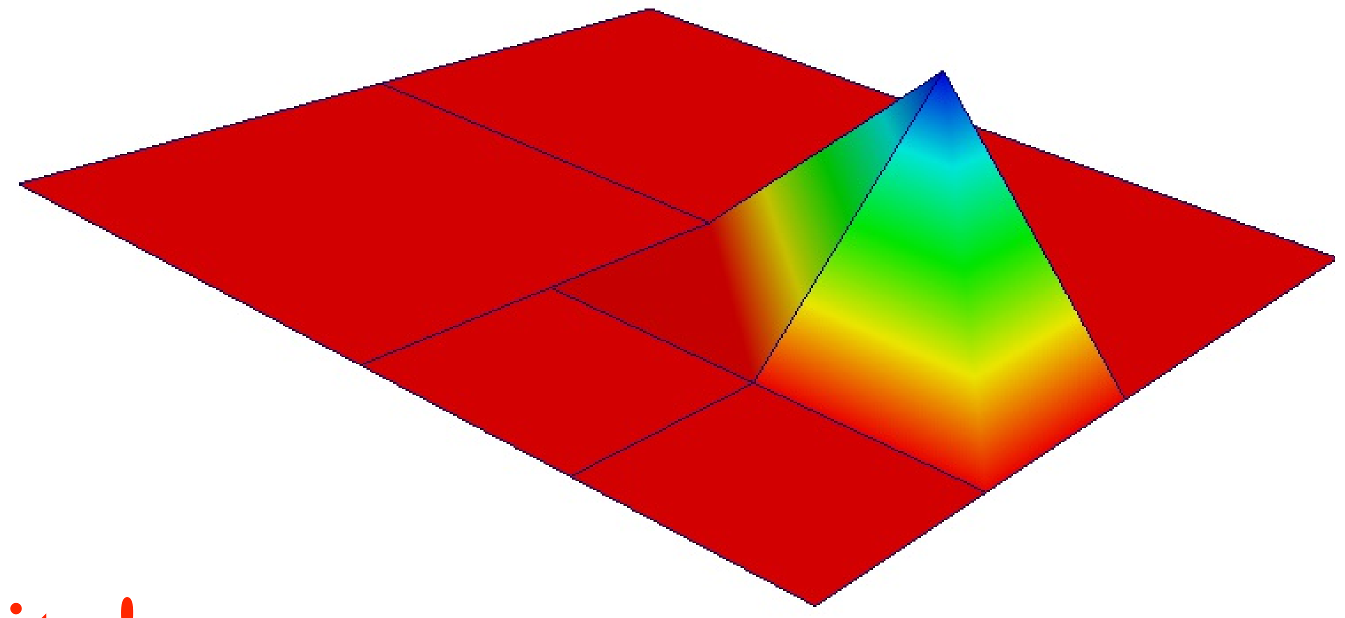
$N_0(\mathbf{x})$ :



$N_2(\mathbf{x})$ :



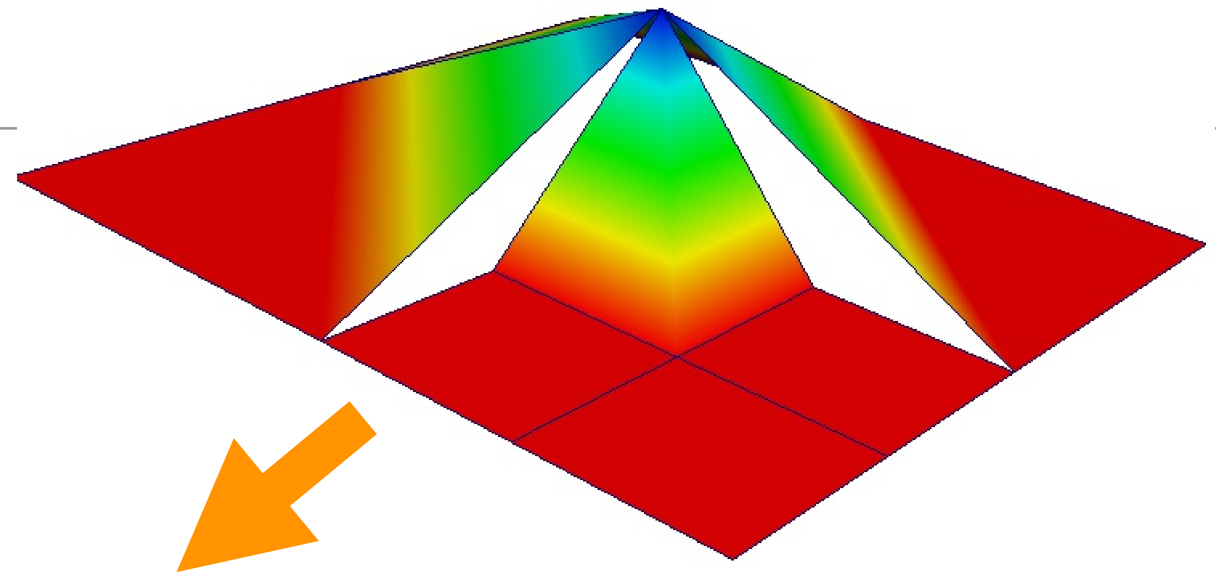
$N_{12}(\mathbf{x})$ :



Solution: introduce constraints to require continuity!

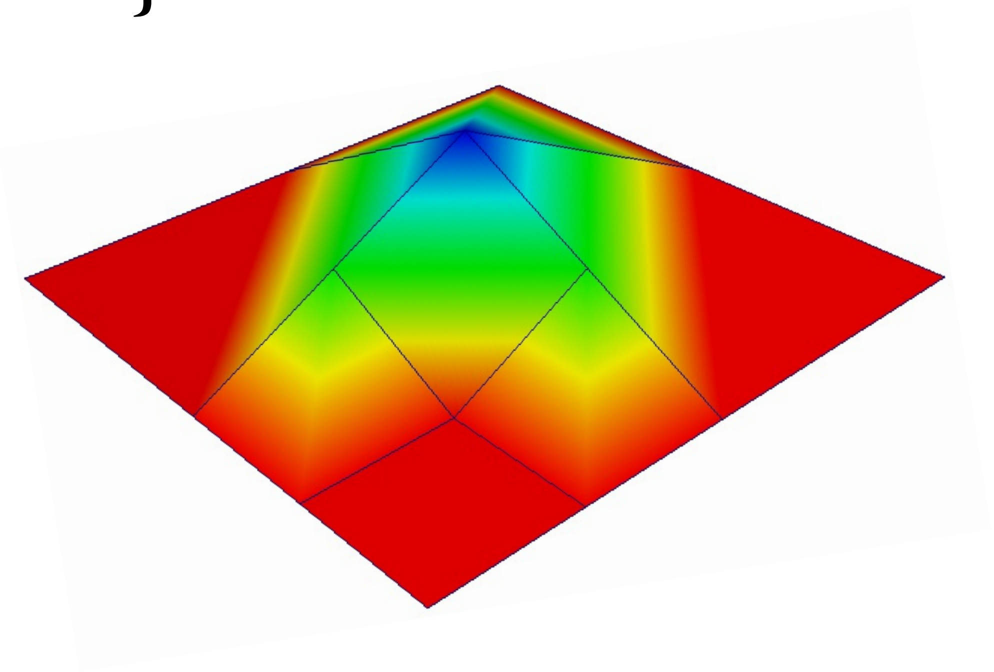
Use standard (possibly globally discontinuous) shape functions, but require continuity of their linear combination

# *Hanging nodes*



$$\mathcal{S}^h = \{u^h = \sum_i u_i N_i(\mathbf{x}) : u^h(\mathbf{x}) \in C^0\}$$

Note, that we encounter



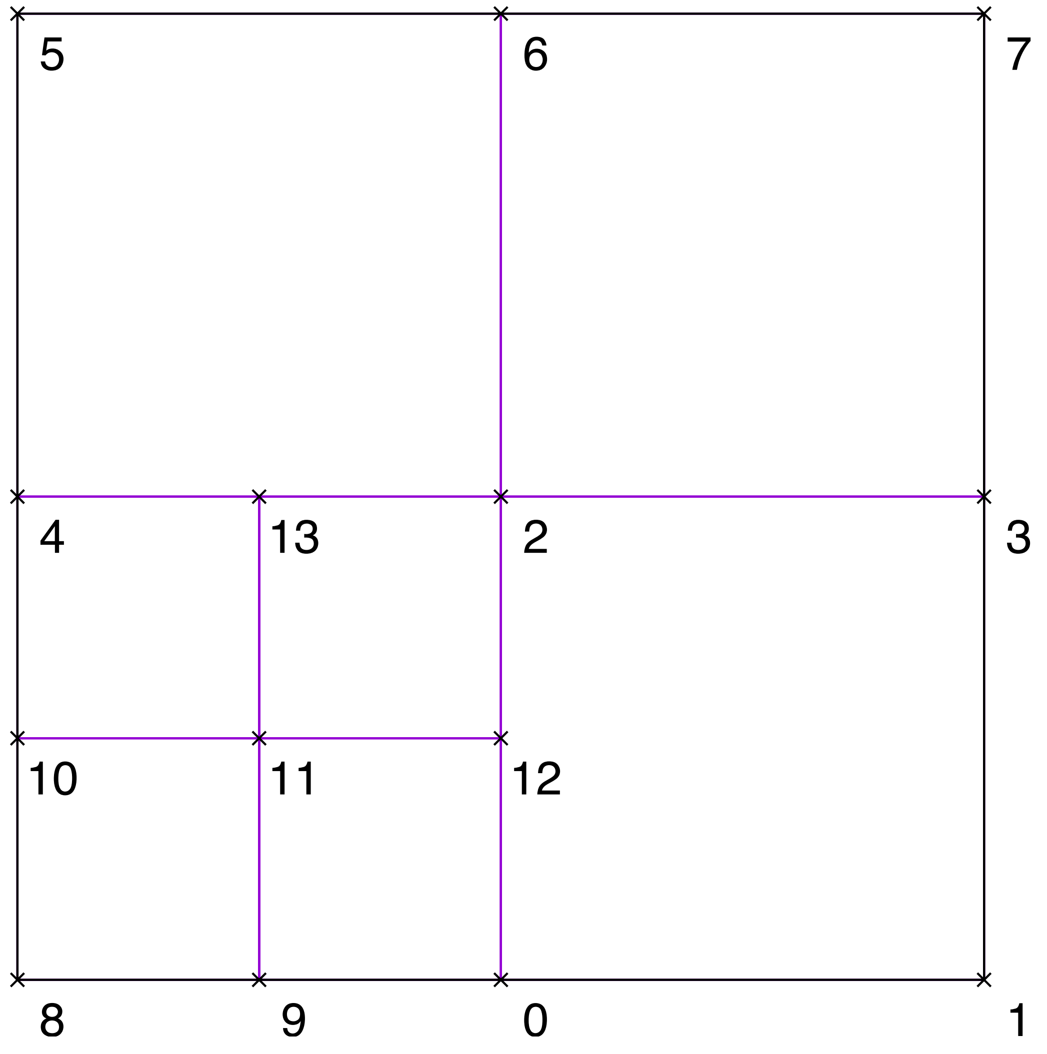
We can make the function continuous by making it

$$u_{12} = \frac{1}{2}u_0 + \frac{1}{2}u_2$$

$$u_{13} = \frac{1}{2}u_2 + \frac{1}{2}u_4$$

define a subset of all DoFs to

$$\mathcal{N}_C \subset \mathcal{N}$$



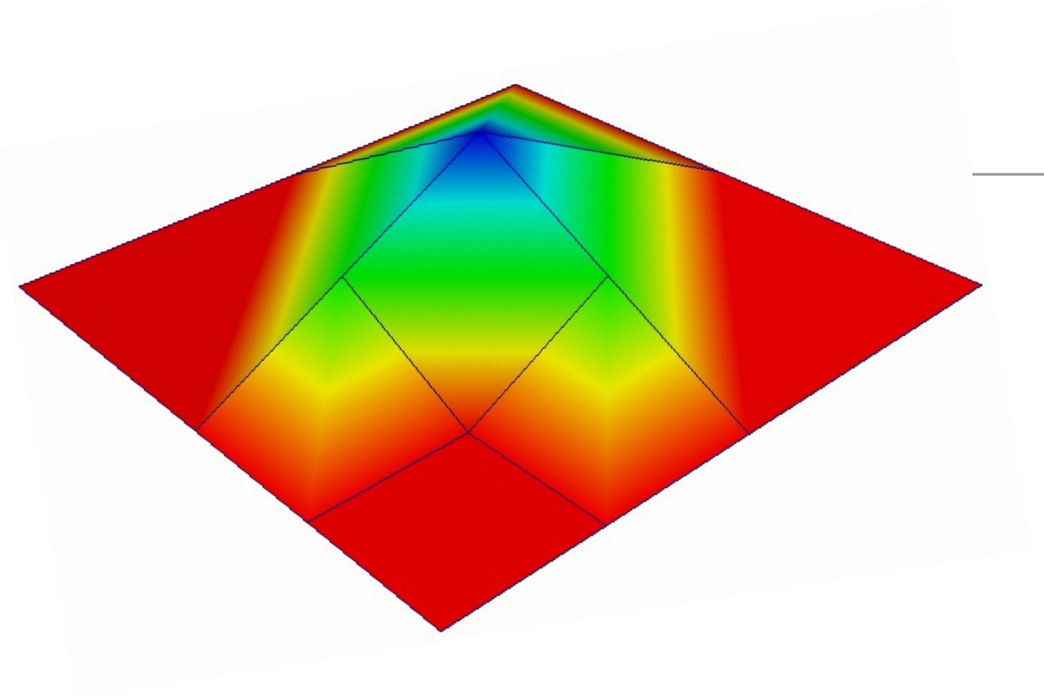

The general

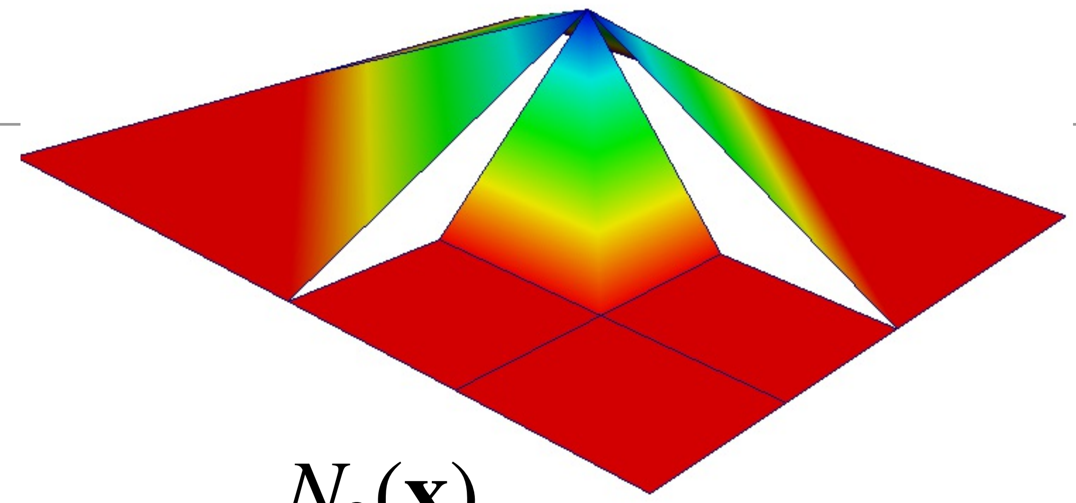
$$u_i = \sum_{j \in \mathcal{N}} c_{ij} u_j + b_i \quad \forall i \in \mathcal{N}_C$$

similar constraints arise from boundary conditions (normal/tangential component) or hp-adaptive FE

# *Condensed shape functions*

The alternative viewpoint is to construct a set of conforming shape functions:

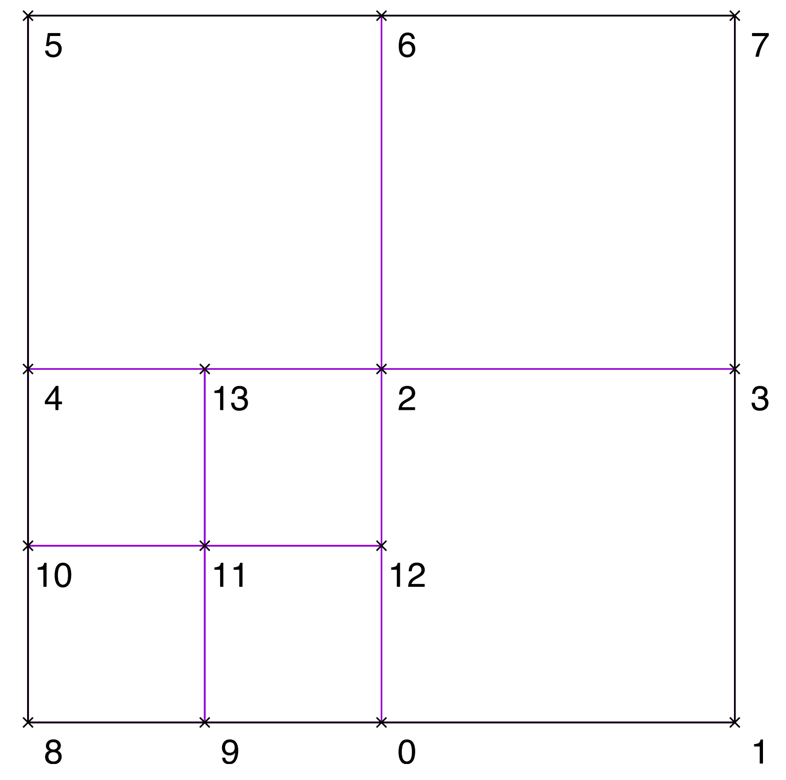




$N_2(\mathbf{x})$

$$\widetilde{N}_2 := N_2 + \frac{1}{2}N_{13} + \frac{1}{2}N_{12}$$

$$\mathcal{S}^h = \{u^h = \sum_{i \in \mathcal{N}/\mathcal{N}_c} u_i \widetilde{N}_i(\mathbf{x})\}$$




$$[\boldsymbol{K}]_{ij} = \begin{cases} a(\widetilde{N}_i, \widetilde{N}_j) & \text{if } i \in \mathcal{N} \setminus \mathcal{N}_c \text{ and } j \in \mathcal{N} \setminus \mathcal{N}_c \\ 1 & \text{if } i \equiv j \text{ and } j \in \mathcal{N}_c \\ 0 & \text{otherwise} \end{cases}$$

$$[\boldsymbol{F}]_i = \begin{cases} (f, \widetilde{N}_i) & \text{if } i \in \mathcal{N} \setminus \mathcal{N}_c \\ 0 & \text{otherwise} \end{cases}$$

The beauty of the approach is that we can assemble local matrix and RHS as

$$\forall i \in \mathcal{N} \backslash \mathcal{N}_c : \quad [\mathbf{F}]_i = (f, \widetilde{N}_i) = (f, N_i + \sum_{j \in \mathcal{N}_c} c_{ji} N_j) = (f, N_i) + \sum_{j \in \mathcal{N}_c} c_{ji}(f, N_j) = \left[\widetilde{\mathbf{F}}\right]_i + \sum_{j \in \mathcal{N}_c} c_{ji} \left[\widetilde{\mathbf{F}}\right]_j$$

## *Using constraints:*

- The beauty of the FEM is that we do exactly the same thing on every cell
- In other words: assembly on cells with hanging nodes should work exactly as on cells without

# Approach 1:

$$\widetilde{\mathcal{S}}^h = \{u^h = \sum_i u_i N_i(x)\}$$

this is not a continuous space, but we may still use it as an intermediate step for matrices!

$$\mathcal{S}^h = \{u^h = \sum_i u_i N_i(x) : u^h(x) \ \in C^0\}$$

Step 1: Build matrix/rhs $\widetilde{\mathbf{K}}, \widetilde{\mathbf{F}}$ with all DoFs as if there were no constraints.

Step 2: Modify $\widetilde{\mathbf{K}}, \widetilde{\mathbf{F}}$ to get $\mathbf{K}, \mathbf{F}$

i.e. eliminate the rows and columns of the matrix that correspond to constrained degrees of freedom

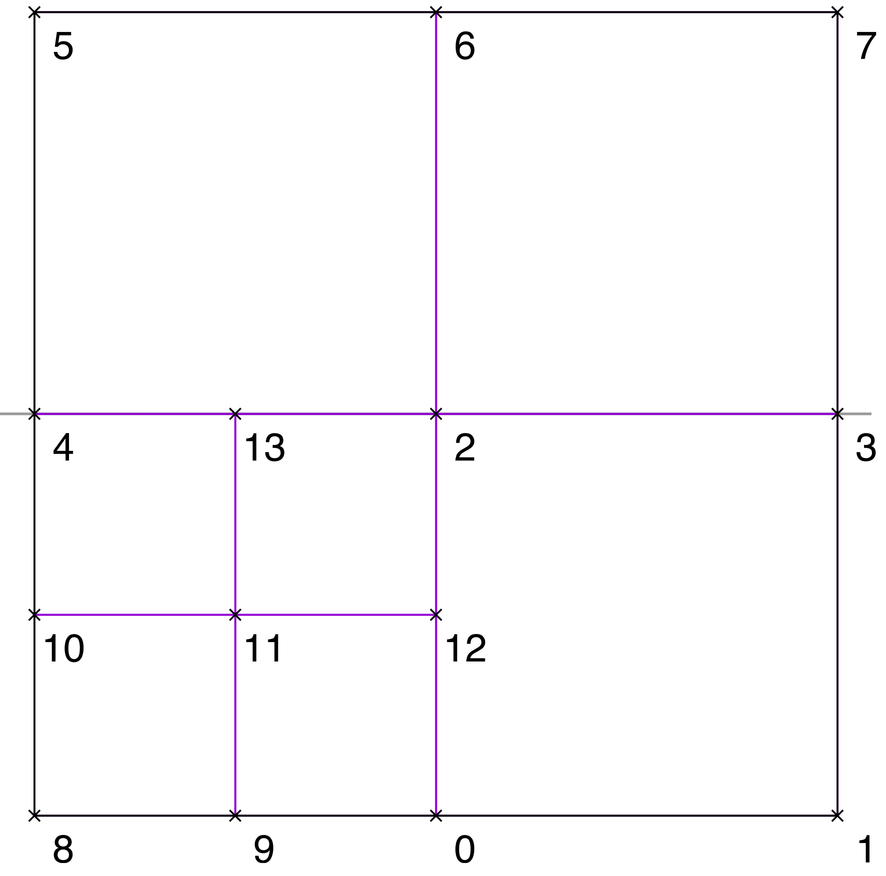
Step 3: Solve $\mathbf{K} \cdot \mathbf{u} = \mathbf{F}$

Step 4: Fill in the constrained components of $\mathbf{u}$ to use $\widetilde{\mathcal{S}}^h$ for evaluation of the field.

Disadvantages: (i) bottleneck for 3d or higher order/hp FEM; (ii) hard to implement in parallel where a

# Approach 1 (example):

$$\begin{bmatrix} u_{12} \\ u_{13} \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_0 \\ u_2 \\ u_4 \end{bmatrix}$$




```
===========================================
Number of active cells: 7
Number of degrees of freedom: 14
=============== constraints ===============
    12 0:  0.5
    12 2:  0.5
    13 2:  0.5
    13 4:  0.5
============== un-condensed ===============
================= matrix ==================
 1.333e+00 -1.667e-01 -1.667e-01 -3.333e-01  0.000e+00                                             -1.667e-01            -3.333e-01 -1.667e-01
-1.667e-01  6.667e-01 -3.333e-01 -1.667e-01
-1.667e-01 -3.333e-01  2.667e+00 -3.333e-01 -1.667e-01 -3.333e-01 -3.333e-01 -3.333e-01             0.000e+00  0.000e+00 -3.333e-01 -1.667e-01 -1.667e-01
-3.333e-01 -1.667e-01 -3.333e-01  1.333e+00                       -3.333e-01 -1.667e-01
 0.000e+00            -1.667e-01             1.333e+00 -1.667e-01 -3.333e-01                                  -1.667e-01 -3.333e-01            -1.667e-01
                      -3.333e-01            -1.667e-01  6.667e-01 -1.667e-01
                      -3.333e-01 -3.333e-01 -3.333e-01 -1.667e-01  1.333e+00 -1.667e-01
                      -3.333e-01 -1.667e-01                       -1.667e-01  6.667e-01
                                                                                         6.667e-01 -1.667e-01 -1.667e-01 -3.333e-01
-1.667e-01             0.000e+00                                                        -1.667e-01  1.333e+00 -3.333e-01 -3.333e-01 -3.333e-01
                       0.000e+00            -1.667e-01                                  -1.667e-01 -3.333e-01  1.333e+00 -3.333e-01            -3.333e-01
-3.333e-01            -3.333e-01            -3.333e-01                                  -3.333e-01 -3.333e-01 -3.333e-01  2.667e+00 -3.333e-01 -3.333e-01
-1.667e-01            -1.667e-01                                                                   -3.333e-01            -3.333e-01  1.333e+00 -3.333e-01
                      -1.667e-01            -1.667e-01                                                        -3.333e-01 -3.333e-01 -3.333e-01  1.333e+00

================ condensed ================
================= matrix ==================
 1.500e+00 -1.667e-01 -8.333e-02 -3.333e-01 -8.333e-02                                             -3.333e-01            -5.000e-01  0.000e+00
-1.667e-01  6.667e-01 -3.333e-01 -1.667e-01
-8.333e-02 -3.333e-01  2.833e+00 -3.333e-01 -8.333e-02 -3.333e-01 -3.333e-01 -3.333e-01            -1.667e-01 -1.667e-01 -6.667e-01  0.000e+00  0.000e+00
-3.333e-01 -1.667e-01 -3.333e-01  1.333e+00                       -3.333e-01 -1.667e-01
-8.333e-02            -8.333e-02             1.500e+00 -1.667e-01 -3.333e-01                                  -3.333e-01 -5.000e-01             0.000e+00
                      -3.333e-01            -1.667e-01  6.667e-01 -1.667e-01
                      -3.333e-01 -3.333e-01 -3.333e-01 -1.667e-01  1.333e+00 -1.667e-01
                      -3.333e-01 -1.667e-01                       -1.667e-01  6.667e-01
                                                                                         6.667e-01 -1.667e-01 -1.667e-01 -3.333e-01
-3.333e-01            -1.667e-01                                                        -1.667e-01  1.333e+00 -3.333e-01 -3.333e-01  0.000e+00
                      -1.667e-01            -3.333e-01                                  -1.667e-01 -3.333e-01  1.333e+00 -3.333e-01             0.000e+00
-5.000e-01            -6.667e-01            -5.000e-01                                  -3.333e-01 -3.333e-01 -3.333e-01  2.667e+00  0.000e+00  0.000e+00
 0.000e+00             0.000e+00                                                                    0.000e+00             0.000e+00  1.333e+00  0.000e+00
                       0.000e+00             0.000e+00                                                         0.000e+00  0.000e+00  0.000e+00  1.333e+00
```

# Approach 2:

$$\widetilde{\mathcal{S}}^h = \{u^h = \sum_i u_i N_i(x)\}$$

$$\mathcal{S}^h = \{u^h = \sum_i u_i N_i(x) : u^h(x) \in C^0\}$$

Step 1: Build local matrix/rhs $\widetilde{\mathbf{K}}_K, \widetilde{\mathbf{F}}_K$ with all DoFs as if there were no constraints.

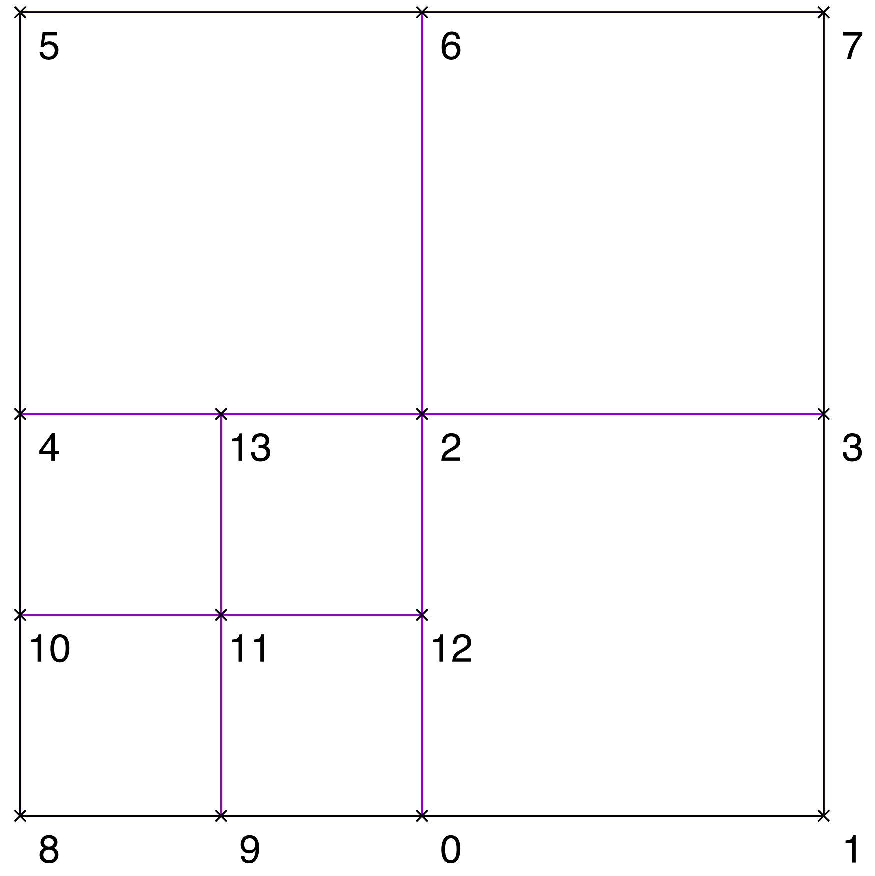
Step 2: Apply constraints during assembly operation (local-to-global) $\mathbf{K}_K, \mathbf{F}_K$

Step 3: Solve $\mathbf{K} \cdot \mathbf{u} = \mathbf{F}$

Step 4: Fill in the constrained components of $\mathbf{u}$ to use $\widetilde{\mathcal{S}}^h$ for evaluation of the field.

# *Approach 2 (example):*

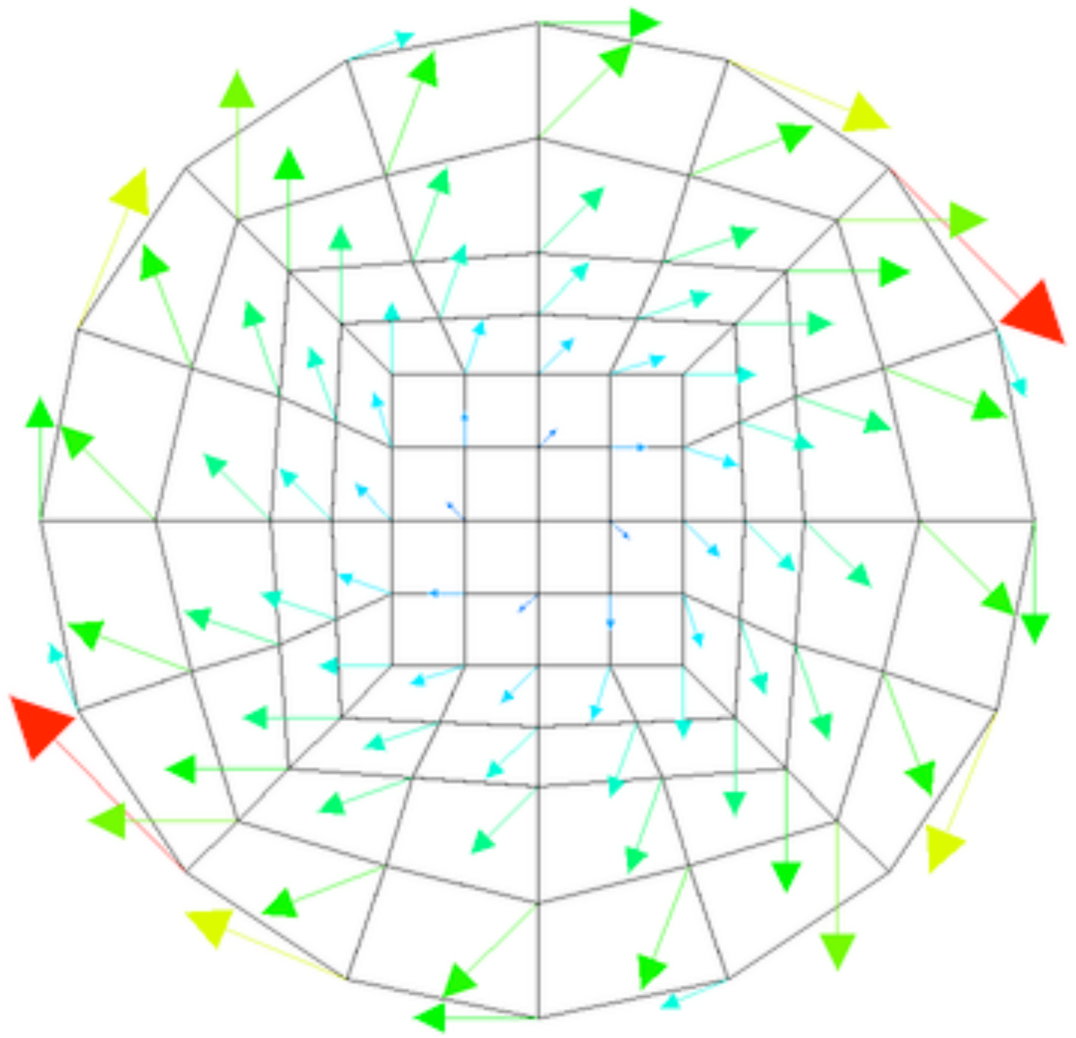
$$\begin{bmatrix} u_{12} \\ u_{13} \end{bmatrix} = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_0 \\ u_2 \\ u_4 \end{bmatrix}$$




```
===========================================
Number of active cells: 7
Number of degrees of freedom: 14
=============== constraints ===============
    12 0:  0.5
    12 2:  0.5
    13 2:  0.5
    13 4:  0.5
================ condensed ================
================= matrix ==================
  1.500e+00 -1.667e-01 -8.333e-02 -3.333e-01 -8.333e-02                                             -3.333e-01            -5.000e-01  0.000e+00
 -1.667e-01  6.667e-01 -3.333e-01 -1.667e-01
 -8.333e-02 -3.333e-01  2.833e+00 -3.333e-01 -8.333e-02 -3.333e-01 -3.333e-01 -3.333e-01            -1.667e-01 -1.667e-01 -6.667e-01  0.000e+00  0.000e+00
 -3.333e-01 -1.667e-01 -3.333e-01  1.333e+00                       -3.333e-01 -1.667e-01
 -8.333e-02            -8.333e-02             1.500e+00 -1.667e-01 -3.333e-01                                  -3.333e-01 -5.000e-01             0.000e+00
                       -3.333e-01            -1.667e-01  6.667e-01 -1.667e-01
                       -3.333e-01 -3.333e-01 -3.333e-01 -1.667e-01  1.333e+00 -1.667e-01
                       -3.333e-01 -1.667e-01                       -1.667e-01  6.667e-01
                                                                                         6.667e-01 -1.667e-01 -1.667e-01 -3.333e-01
 -3.333e-01            -1.667e-01                                                       -1.667e-01  1.333e+00 -3.333e-01 -3.333e-01  0.000e+00
                       -1.667e-01            -3.333e-01                                 -1.667e-01 -3.333e-01  1.333e+00 -3.333e-01             0.000e+00
 -5.000e-01            -6.667e-01            -5.000e-01                                 -3.333e-01 -3.333e-01 -3.333e-01  2.667e+00  0.000e+00  0.000e+00
  0.000e+00             0.000e+00                                                                   0.000e+00             0.000e+00  1.333e+00  0.000e+00
                        0.000e+00             0.000e+00                                                        0.000e+00  0.000e+00  0.000e+00  1.333e+00
```

# *Applying constraints: the AffineConstraints class*

- This class is used for
  - Hanging nodes
  - Dirichlet and periodic constraints
  - Other constraints
- Linear constraints of the the form $u_C = Cu_O + b$



# Applying constraints: the AffineConstraints class

- System setup
  - Hanging node constraints created using
    `DoFTools::make_hanging_node_constraints()`
  - Will also use for boundary values from now on:
    `VectorTools::interpolate_boundary_values(..., constraints);`
  - Need different SparsityPattern creator
    `DoFTools::make_sparsity_pattern (..., constraints, …)`
    - Can remove constraints from linear system
      `DoFTools::make_sparsity_pattern (..., constraints, / *keep_constrained_dofs = * / false)`
  - Sort, rearrange, optimise constraints
    `constraints.close()`

# *Applying constraints: the AffineConstraints class*

- Assembly
  - Assemble local matrix and vector as normal
  - Eliminate while transferring to global matrix:

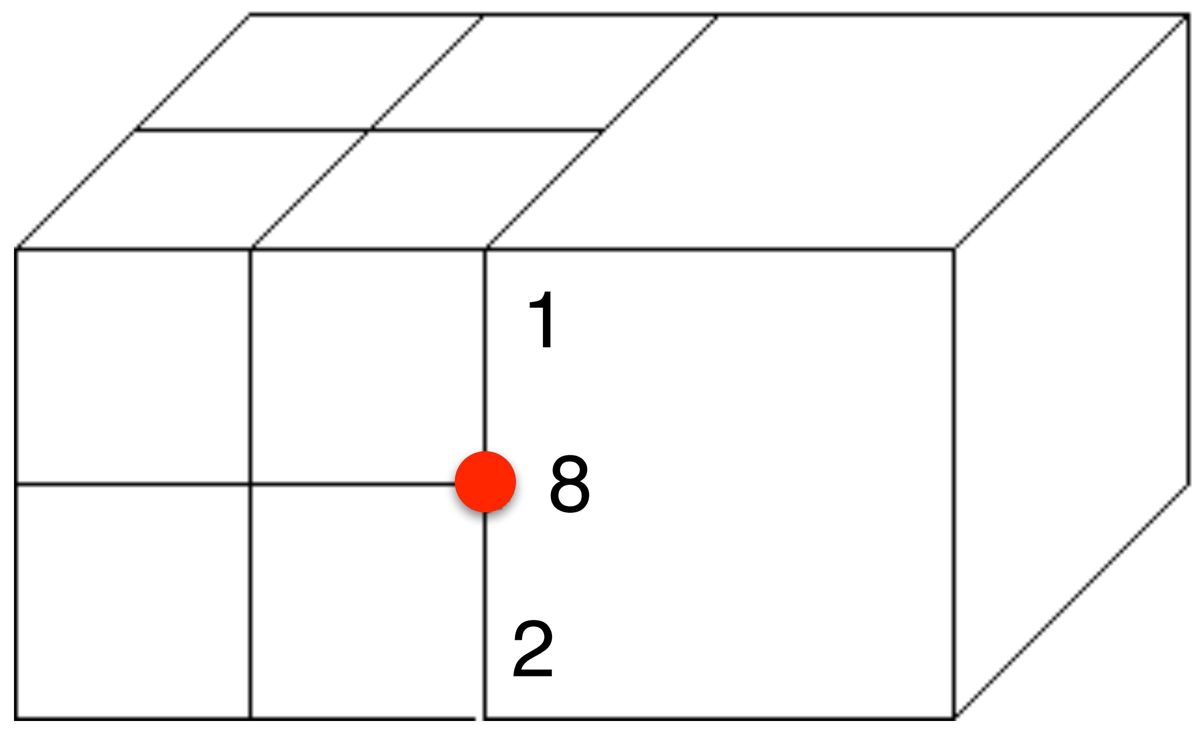
```
constraints.distribute_local_to_global (
  cell_matrix, cell_rhs,
  local_dof_indices,
  system_matrix, system_rhs);
```

  - Solve and then set all constraint values correctly:

```
ConstraintMatrix::distribute(...)
```

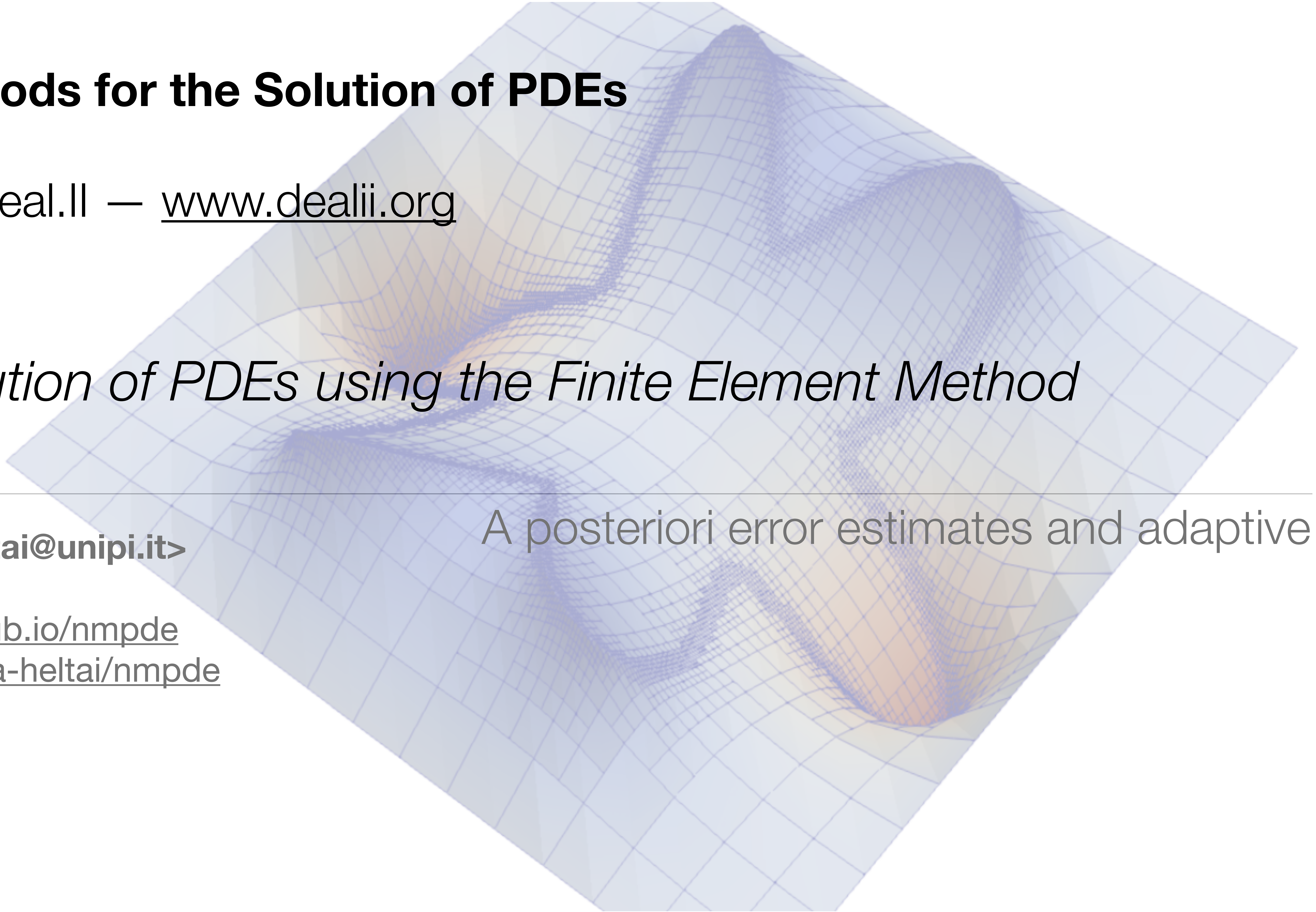
# *Applying constraints: Conflicts*

- When writing into a AffineConstraints, existing constraints are not overwritten.
- Can merge constraints together:
```
constraints.merge (other_constraints,
MergeConflictBehavior::left_object_wins);
```
- Which is right? $u_8 = \bar{u}$ or $u_8 = \frac{1}{2}\left[u_1 + u_2\right]$
- Careful on loops: $u_1 = u_2 \; ; \; u_2 = u_3 \; ; \; u_3 = u_1$



# Numerical Methods for the Solution of PDEs

Laboratory with deal.II — www.dealii.org



*Numerical Solution of PDEs using the Finite Element Method*

A posteriori error estimates and adaptive me

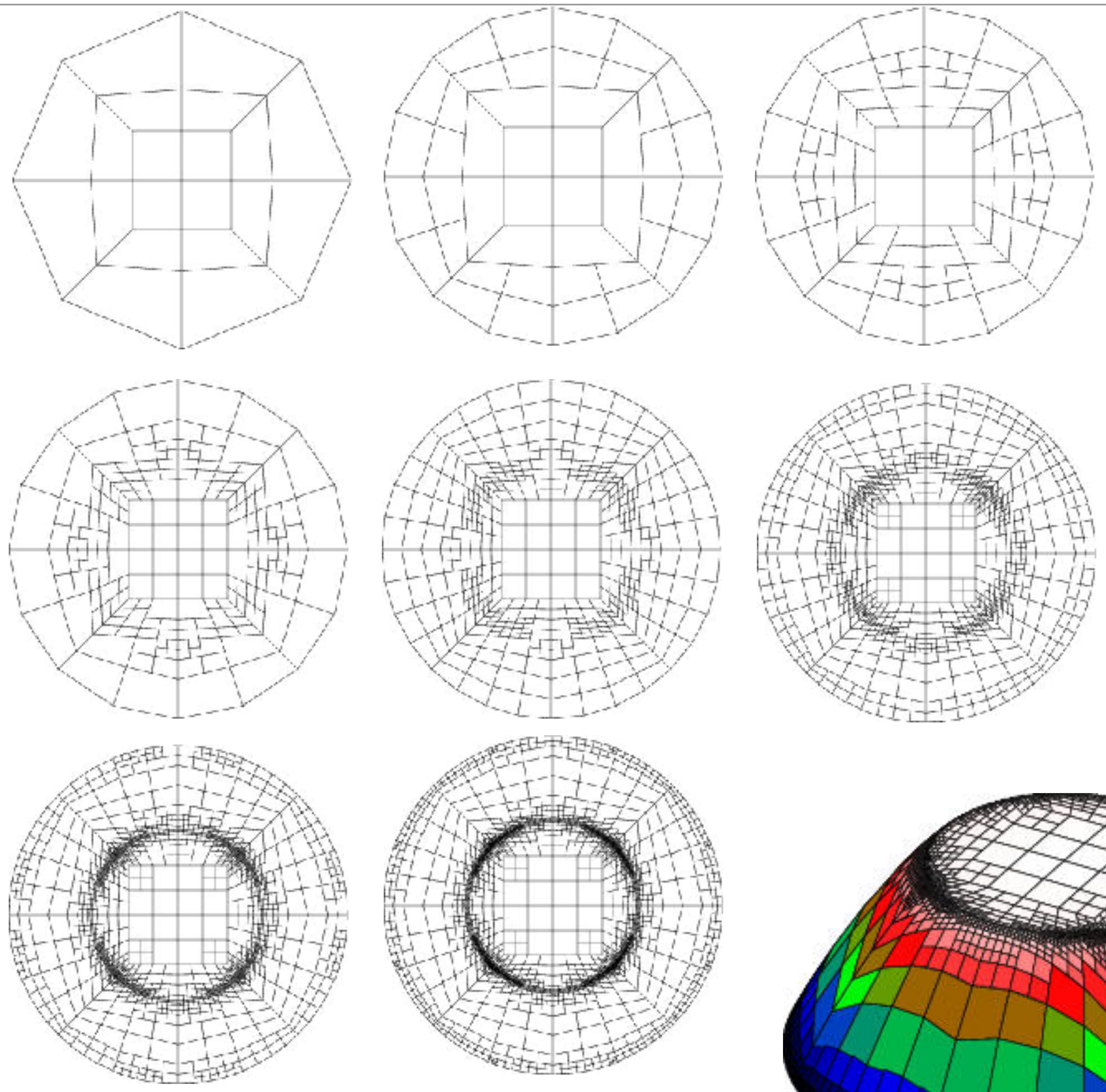
**Luca Heltai <luca.heltai@unipi.it>**

https://luca-heltai.github.io/nmpde
https://github.com/luca-heltai/nmpde

# *Adaptive mesh refinement*

- Typical steps to perform adaptivity
  - Solve (non-)linear system
  - Estimate error
  - Mark cells
  - Refine/coarsen
  - Interpolate original solution to new mesh

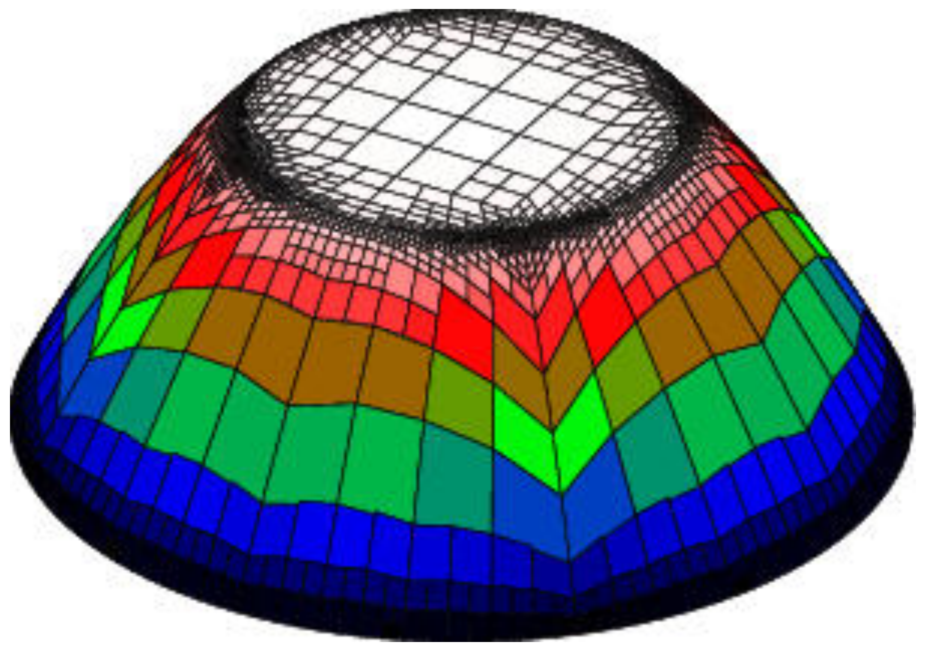
Need an error indicator on each cell without knowing the exact solution. $\eta_K$



$$\nabla \cdot a(\boldsymbol{x}) \nabla u(\boldsymbol{x}) = 1 \quad \text{in } \Omega$$

$$u = 0 \quad \text{on } \partial\Omega$$

$$a(\boldsymbol{x}) = \begin{cases} 20 & \text{if } |\boldsymbol{x}| < 0.5 \\ 1 & \text{otherwise} \end{cases}$$



# *Adaptive mesh refinement*

Error estimate for Q1/P1 elements applied to Laplace problem:

$$\|u - u^h\|_{H^1} \equiv \|e\|_{H^1} \leq C\, h_{max}\, \|u\|_{H^2}$$

this error depends on the largest element size and the global norm of the solution.

$$\|u\|^2_{H^2(K)} := \int_K u^2 + |\nabla u|^2 + |\nabla^2 u|^2$$

$$|u|^2_{H^2(K)} := \int_K |\nabla^2 u|^2$$

To reduce error (increase accuracy) one can refine the mesh size.

more precisely...

$$\|e\|^2_{H^1} \leq C^2 \sum_K h_K^2 \, |u|^2_{H^2(K)}$$

Thus one needs to make mesh finer where the local $H^2$ semi-norm is large.

But apart from some special cases we don't know the exact solution $u$ !

Thus we need to create meshes iteratively (adaptively).

Optimal strategy is to equilibrate the error $e_K := C\, h_K\, |u|_{H^2(K)}$

That is, we want to choose $h_K \sim \dfrac{1}{|u|_{H^2(K)}}$

# *a-posteriori error estimation*

$$||e||^2_{H^1(\Omega)} \leq C \sum_K e_K^2$$

$$e_K = h_K ||\nabla^2 u||_K$$

cell-wise error indicators

(wrong) idea:

$$e_K \approx h_K ||\nabla^2 u^h||_K$$

will not work as linear elements have zero second derivates within the element and first derivatives have jumps on the interfaces

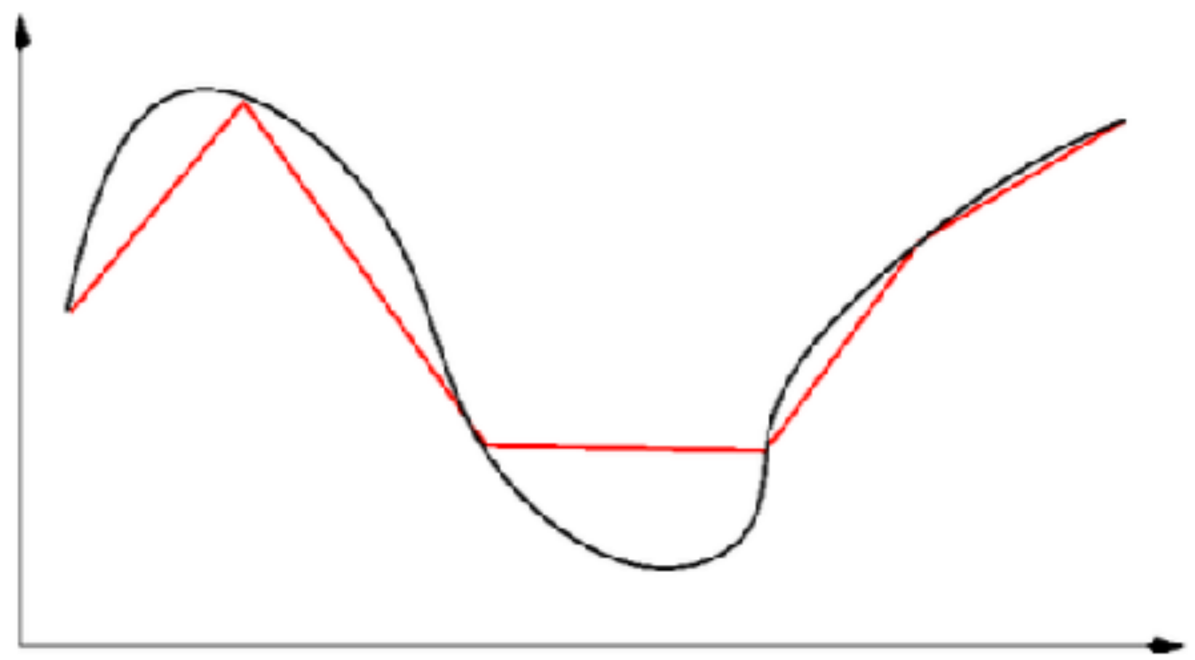
a better idea (in 1D) to approximate second derivatives at interface i:

$$\nabla^2 u \approx \frac{\nabla u^h(x^+) - \nabla u^h(x^-)}{h} =: \frac{[\![\nabla u^h]\!]_i}{h}$$

can generalize to:

$$||\nabla^2 u||^2_K \approx \sum_{i \in \partial K} \frac{[\![\nabla u^h]\!]_i^2}{h}$$

use jump in gradient as an indicator of the second derivative at vertices



(courtesy of Wolfgang Bangerth)

# *a-posteriori error estimation*

As a result, the simplest and most widely used Kelly error **indicator** in 2D/3D follows:

$$e_K^2 = h_K^2 ||\nabla^2 u||_K^2 \approx h_K \int_{\partial K} |[\![\boldsymbol{\nabla} u \cdot \boldsymbol{n}]\!]|^2 \mathrm{ds} =: \eta_{\mathrm{K}}^2$$

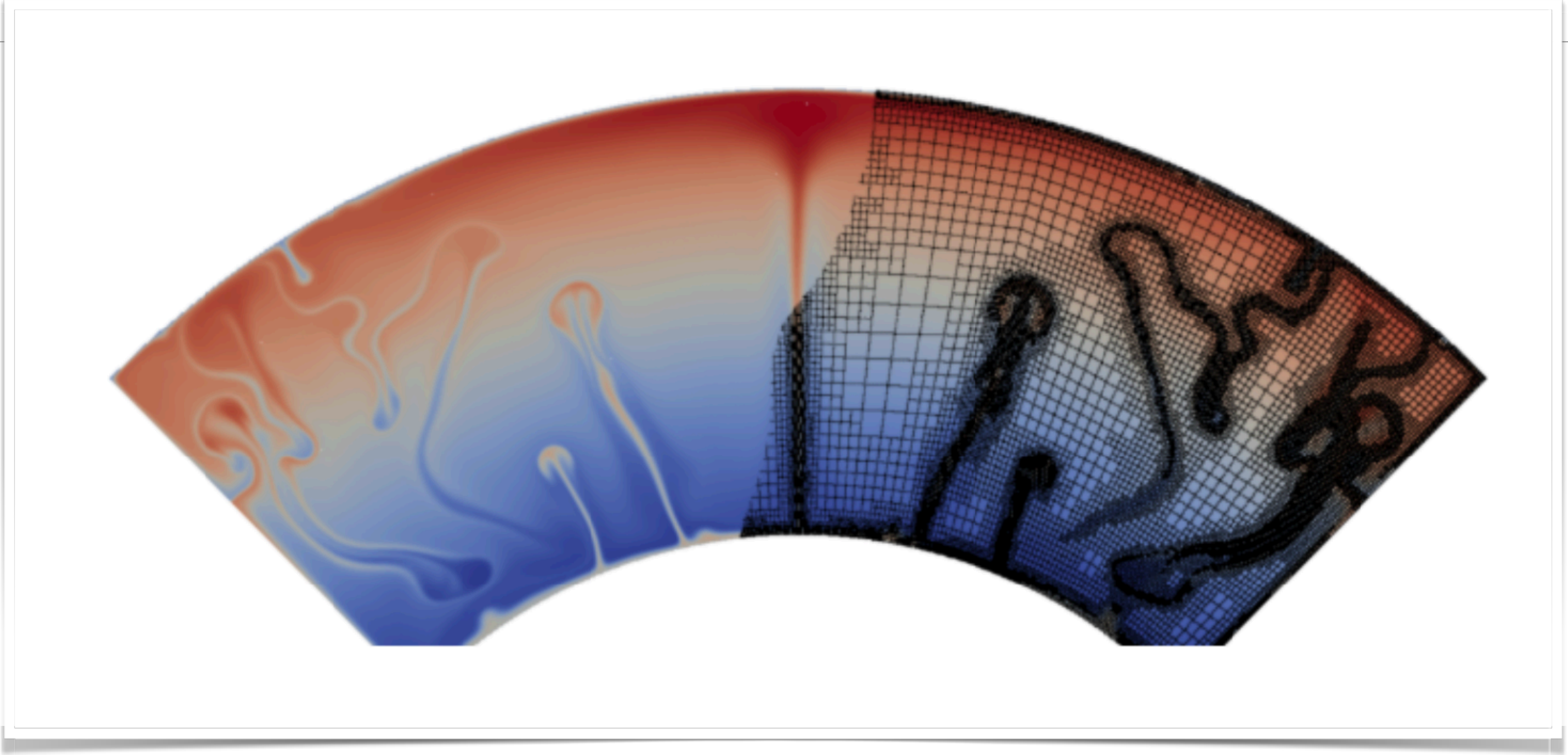
For the Laplace equation, **Kelly, de Gago, Zienkiewicz, Babushka (1983)** proved that

$$||\nabla \left[u - u^h\right]||^2 \leq C \sum_K \eta_K^2$$ a-posteriori error estimator (*)

Note 1:

"**estimator**" is always a <u>proven upper bound of error</u> (*), whereas "**indicator**" is our best guess of error per cell which may not be an upper bound in the sense (*), but may still work well for considered equations and/or FE space.

# *Adaptive mesh refinement*



Refine where things are happening!

# *Basic AFEM algorithm*

- SOLVE-ESTIMATE-MARK-REFINE
  - On the current mesh, solve the problem
  - Estimate the error per cell (Exact, Kelly, Residual, etc.)
  - Mark cells according to given criterion (estimator is greater than a tolerance, or fraction of cells with largest error, or …)
  - Refine the marked cells
- Repeat until tolerance met, or max number of cycles

# *deal.II classes*

- Error estimate is problem dependent:
  - Approximate gradient jumps: `KellyErrorEstimator` class
  - Approximate local norm of gradient: `DerivativeApproximation` class
  - … or something else
- Cell marking strategy:
  - `GridRefinement::refine_and_coarsen_fixed_number(...)`
  - `GridRefinement::refine_and_coarsen_fixed_fraction(...)`
  - `GridRefinement::refine_and_coarsen_optimize(...)`
- Refine/coarsen grid: `triangulation.execute_coarsening_and_refinement ()`
- Transferring the solution: `SolutionTransfer` class (discussed later)