deal.II Users and Developers Training

March 21 – 24 2016

Timo Heister (heister@clemson.edu)
Luca Heltai (luca.heltai@sissa.it)

Adaptive mesh refinement (AMR)

“Traditional” error estimates for Q1/P1 elements applied to the Laplace equation look like this:

$$\| e \|_{H^1} \leq C h_{\max} \| u \|_{H^2} \qquad \text{or equivalently:} \quad \| e \|_{H^1}^2 \leq C^2 h_{\max}^2 \| u \|_{H^2}^2$$

This implies that the error is dominated by the *largest* cell diameter and the (global) H^2 norm of the exact solution.

To reduce the error, this suggests:

- Global mesh refinement
- Nothing can be done if a solution has a large H^2 norm

Adaptive mesh refinement (AMR)

However, a closer analysis shows that the error is really:

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

In other words: To reduce the error, we *only* need to make the mesh fine where the local H^2 norm is large!

Adaptive mesh refinement (AMR)

Note: The optimal strategy to minimize the error while keeping the problem as small as possible is to *equilibrate* the local contributions

$$e_K = C h_K \| u \|_{H^2(K)}$$

That is, we want to choose

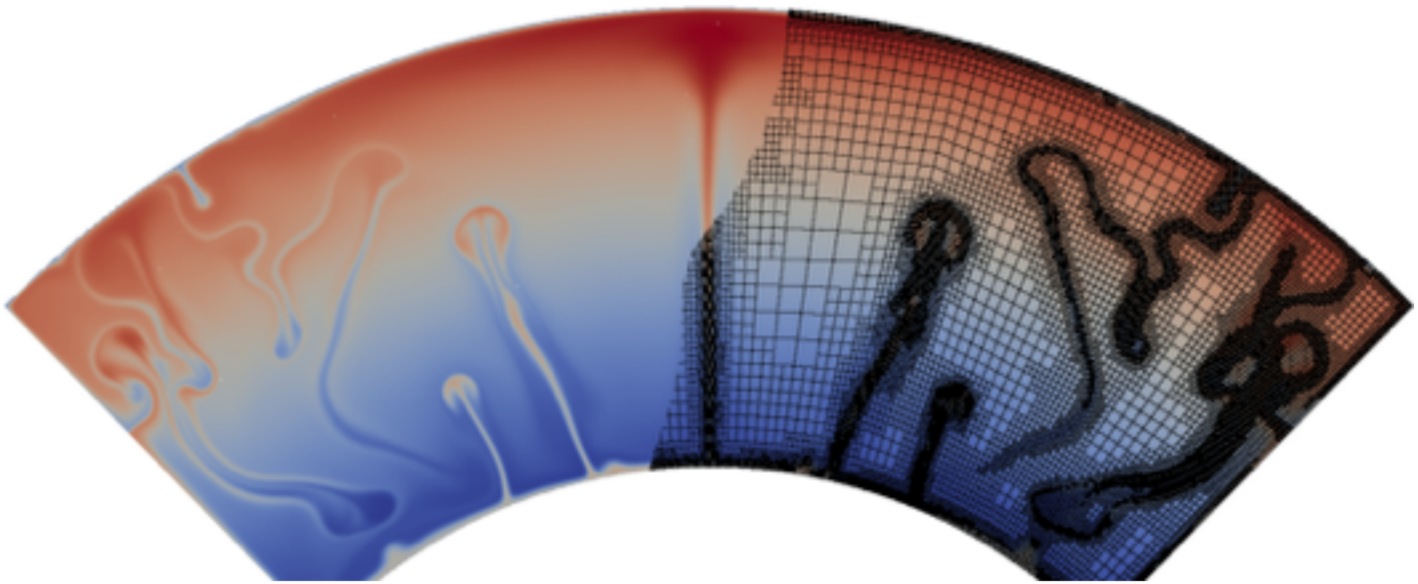
$$h_K \propto \frac{1}{\| u \|_{H^2(K)}}$$

In practice: Exact errors are unknown. Thus, use a local *indicator* of the error η_K and choose h_K so that

$$\sum_K \eta_K = \mathrm{tol}$$

Adaptive mesh refinement (AMR)

Example:



Refine only where “something is happening” (i.e., locally the second derivative of the solution is large).

Adaptive mesh refinement (AMR)

Question: How can we create such meshes?

Answer 1: Except for special cases it is not possible to generate them right away because we do not know the exact solution.

Answer 2: But it can be done iteratively.

Adaptive mesh refinement (AMR)

Basic algorithm:

1. Start with a coarse mesh
2. Solve on this mesh
3. Compute error indicator for every cell based on numerical solution
4. If overall error is less than *tol* then STOP
5. Mark a fraction of the cells with largest error
6. Refine marked cells to get new mesh
7. Start over at 2.

Note: This is often referred to as the SOLVE-MARK-REFINE cycle.

Refining triangular meshes

Refining *triangular* meshes is relatively simple.

There are three widely used options:

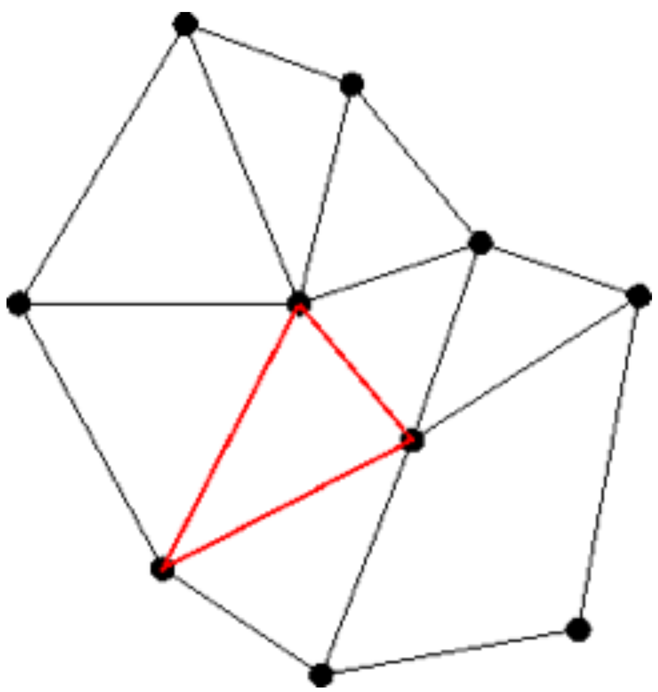
- *De novo* generation of a non-uniform mesh
- Longest edge refinement
- Red-green refinement

Note 1: Refining tetrahedra in 3d works in analogous ways.

Note 2: There are also other strategies.

Longest edge refinement

Consider this mesh and a marked cell:

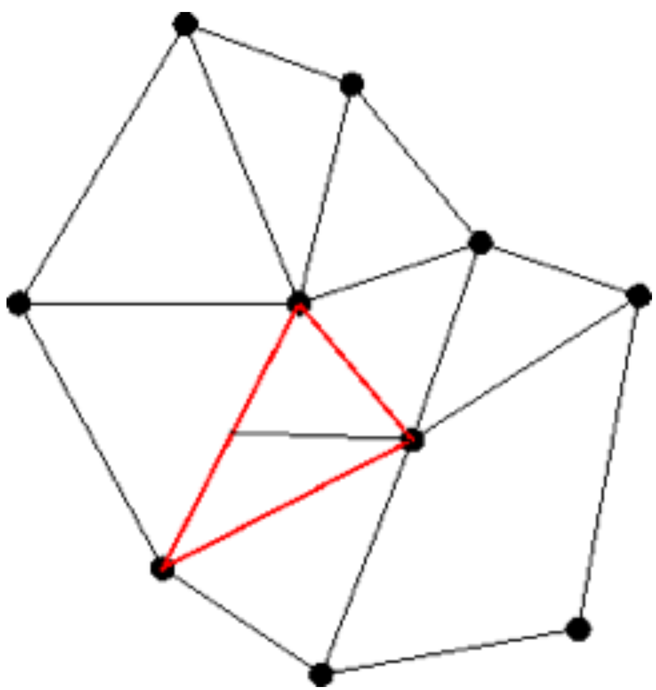


Algorithm:

– Add edge from midpoint of longest edge to oppos. vertex

Longest edge refinement

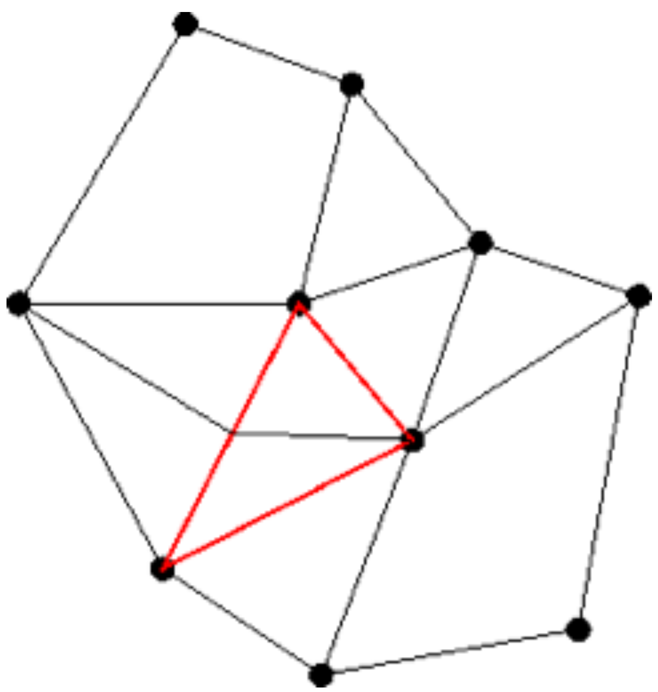
Consider this mesh and a marked cell:



Algorithm:
- Add edge from midpoint of longest edge to oppos. vertex
- If this is also the longest edge of neighboring cell, add edge to opposite vertex

Longest edge refinement

Consider this mesh and a marked cell:



Algorithm:

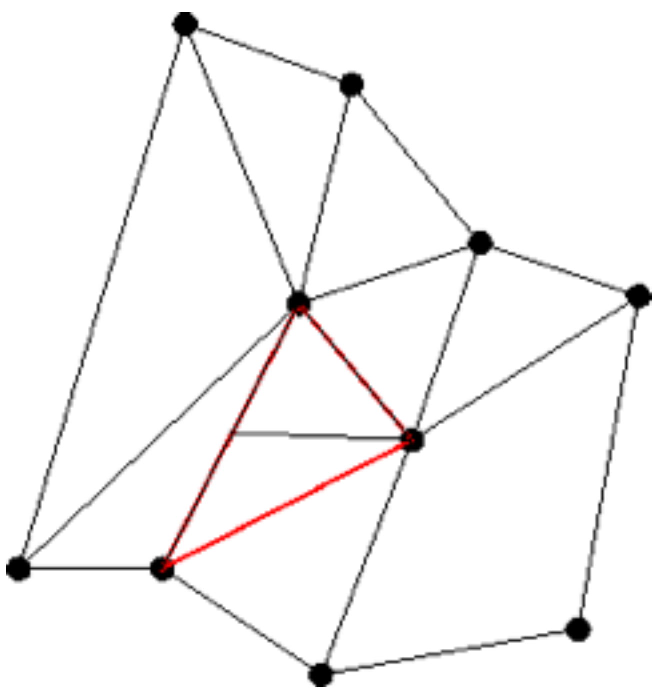
– Add edge from midpoint of longest edge to oppos. vertex

– If this is also the longest edge of neighboring cell, add edge to opposite vertex

– DONE

Longest edge refinement

Consider this variant:



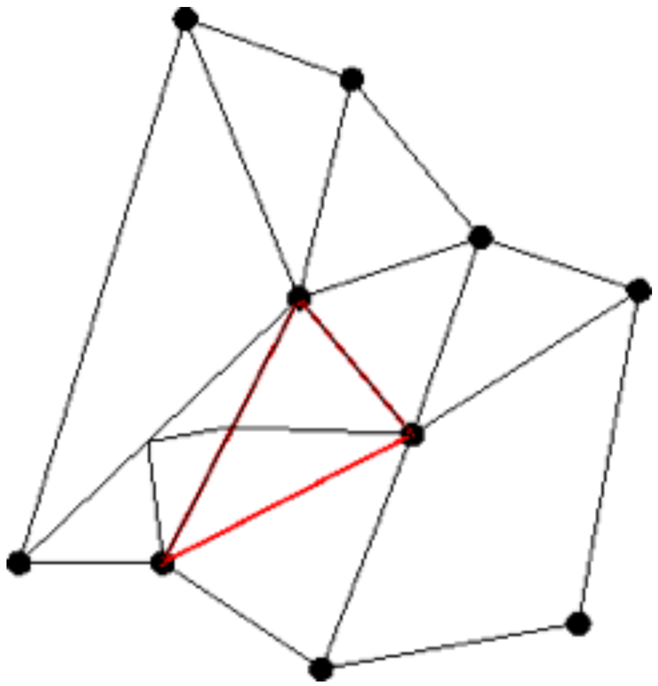
Algorithm:

– Add edge from midpoint of longest edge to oppos. vertex

– Connecting to opposite vertex of neighbor cell would yield distorted cell!

Longest edge refinement

Consider this variant:

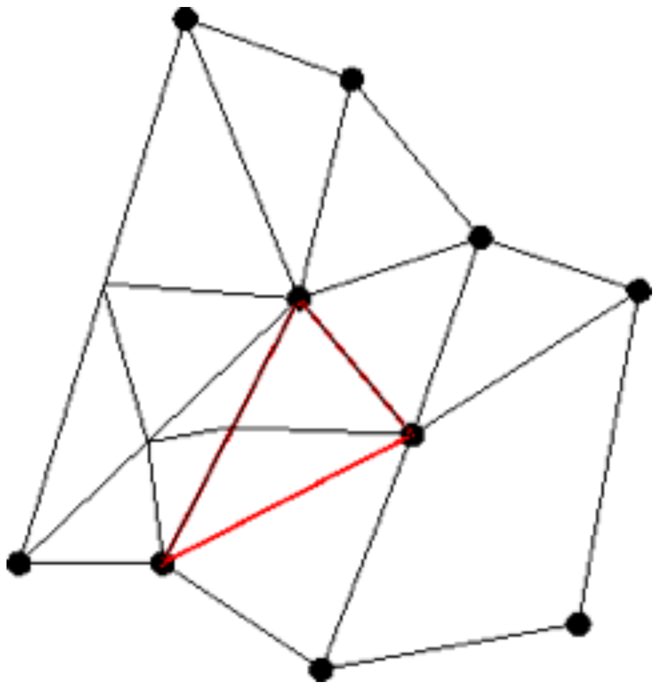


Algorithm:

– Add edge from midpoint of longest edge to oppos. vertex
– If this is *not* the longest edge of neighboring cell, add edges to midpoint of longest edge *and* from there to the opposite vertex

Longest edge refinement

Consider this variant:



Algorithm:

- Add edge from midpoint of longest edge to oppos. vertex
- If this is *not* the longest edge of neighboring cell, add edges to midpoint of longest edge *and* from there to the opposite vertex
- repeat

Credits: **Wolfgang Bangerth**

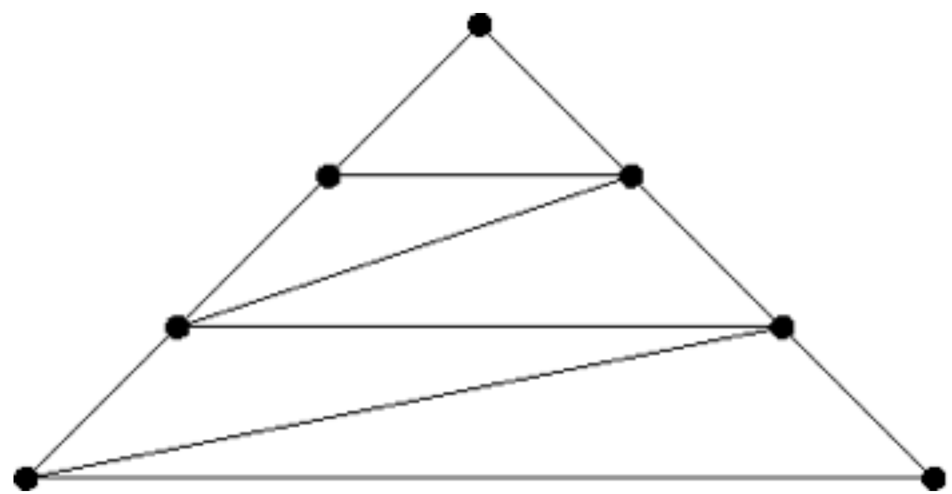
Longest edge refinement

Analysis:

- The algorithm is designed to keep the triangles from degenerating
- However, refinement is not *local*: we may have to refine a set of neighboring elements as well!

Longest edge refinement

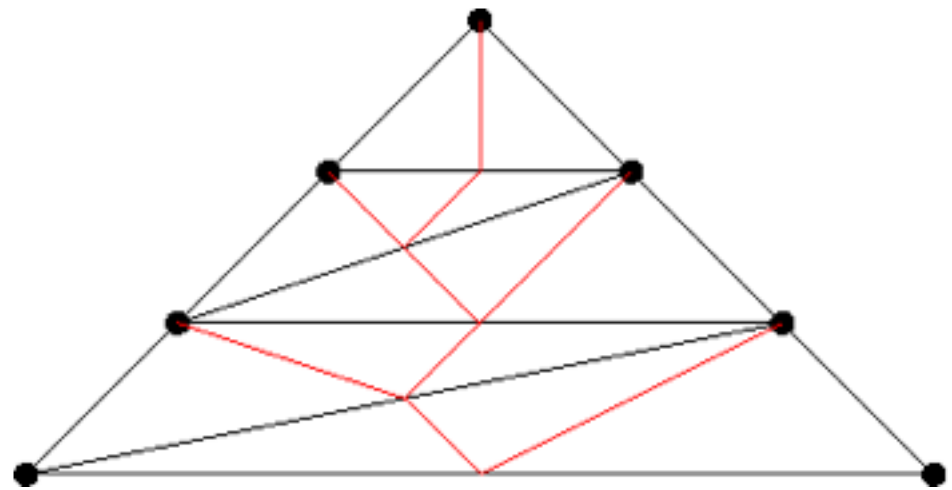
Example of runaway non-local refinement:



Goal: Refine the top-most cell.

Longest edge refinement

Example of runaway non-local refinement:

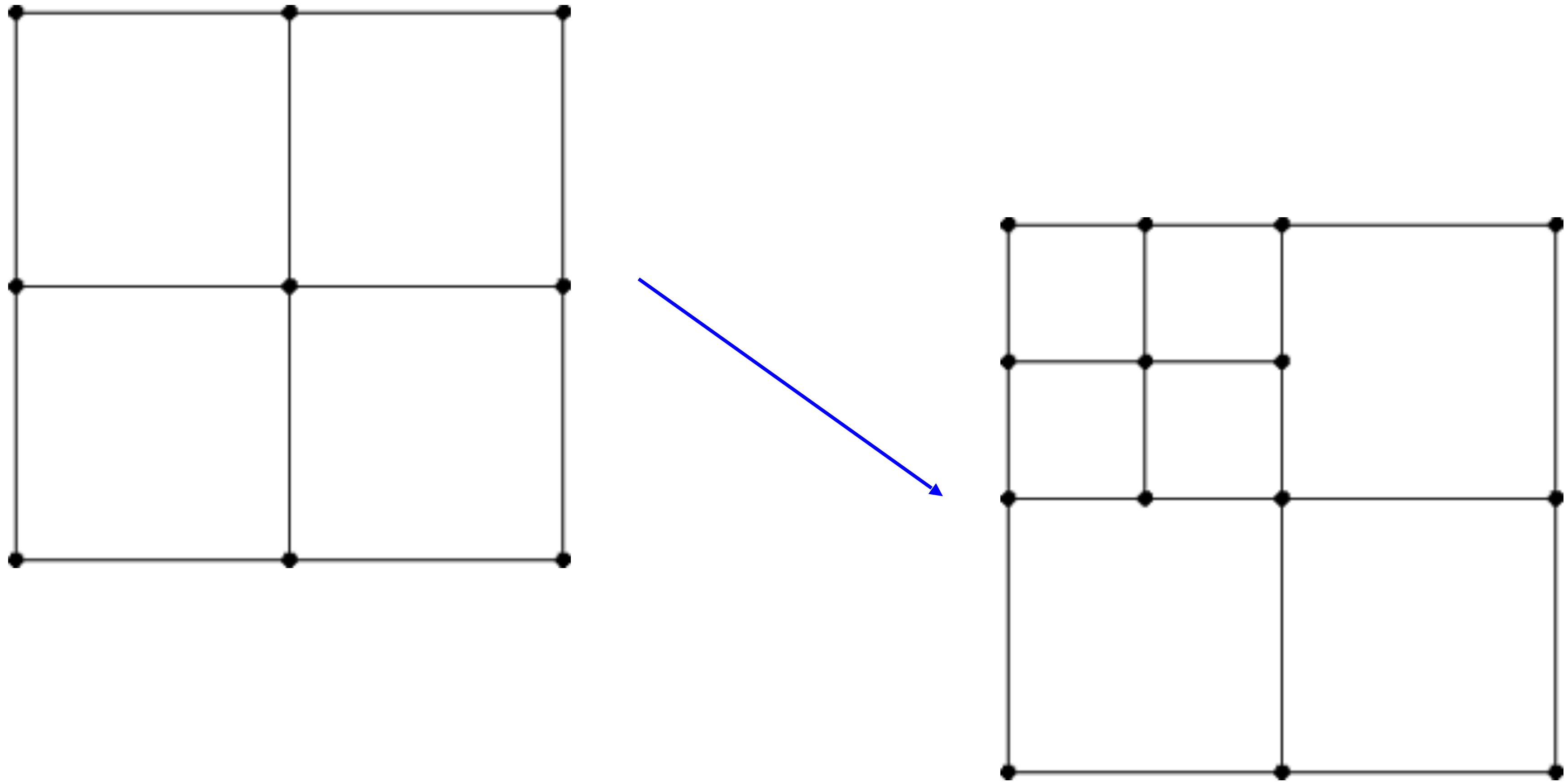


Result: *All* cells are refined!

Note: This situation appears contrived but is quite common in tight corners of difficult geometries. There are even infinite recursions!

Quad/hex refinement

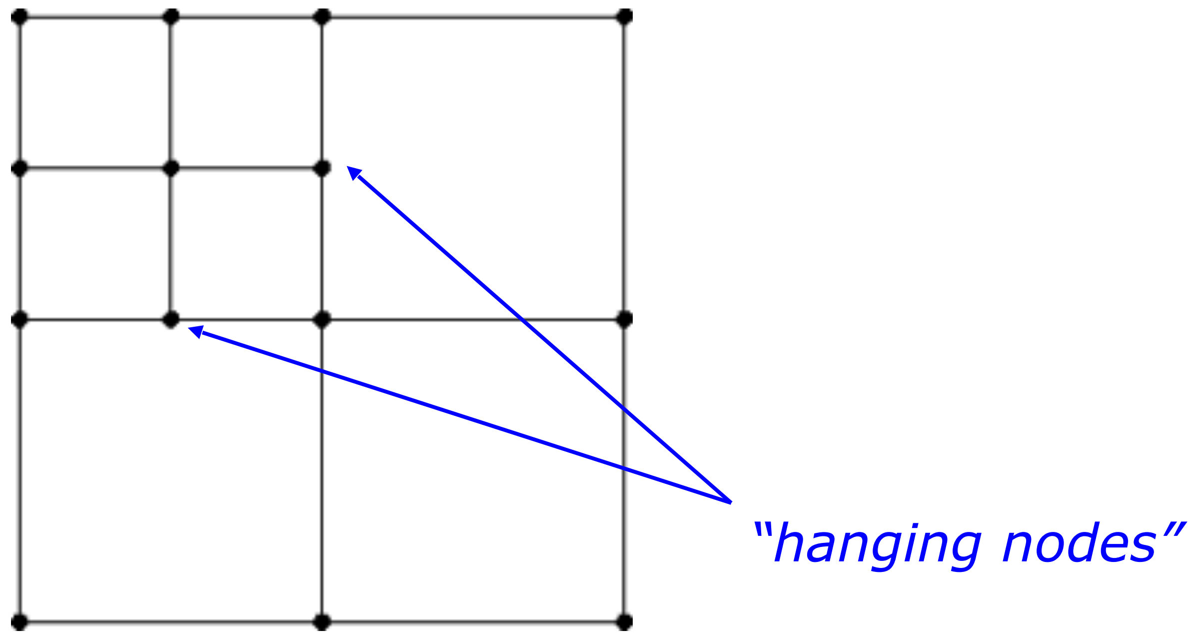
For quads/hexes, the situation is more difficult:



There are no easy options to keep the children of the top right/bottom left cell as quadrilaterals!

Quad/hex refinement

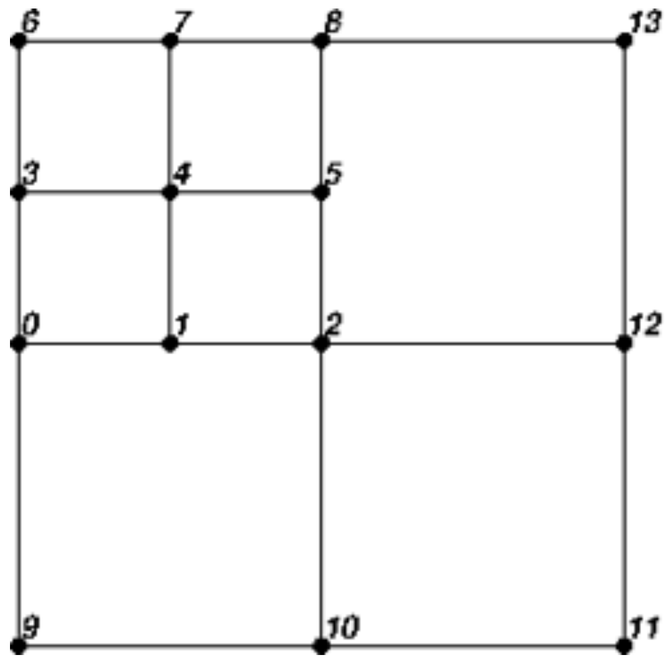
For quads/hexes, the situation is more difficult:



Adaptive quad/hex meshes usually just keep these "hanging nodes" and have other ways to deal with the consequences!

Why constraints?

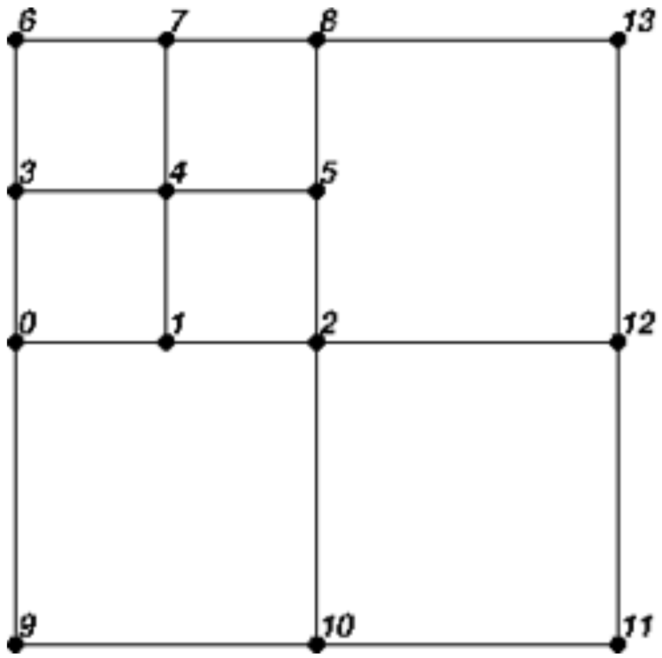
Consider this mesh, Q1 elements, and DoFs as enumerated:

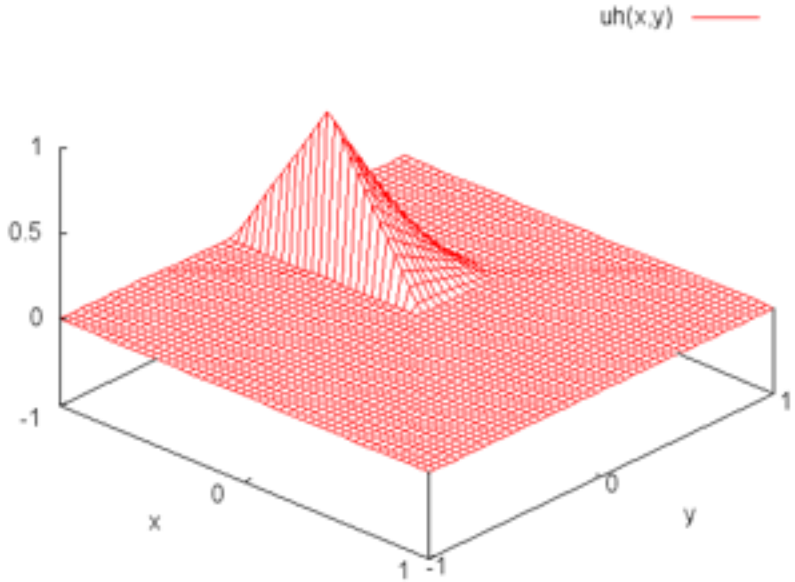


The corresponding space has dimension 14.

Why constraints?

Now consider a solution vector *U=(0,1,0,0,...)* and the function u_h associated with it:





Note: This function is not continuous!

Credits: **Wolfgang Bangerth**

Why constraints?

If our function space V_h has discontinuous functions:

- It is no longer a subspace of the usual H^1
- A bilinear form such as

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v \; dx$$

no longer makes immediate sense

Resolution:

For spaces such as Q_1, we really need to *require* continuity!
We do so through *constraints*.

Why constraints?

Defining V_h via constraints:

- Shape functions are defined on each cell as usual
- Functions in V_h are linear combinations of shape functions
- Functions in V_h are globally continuous

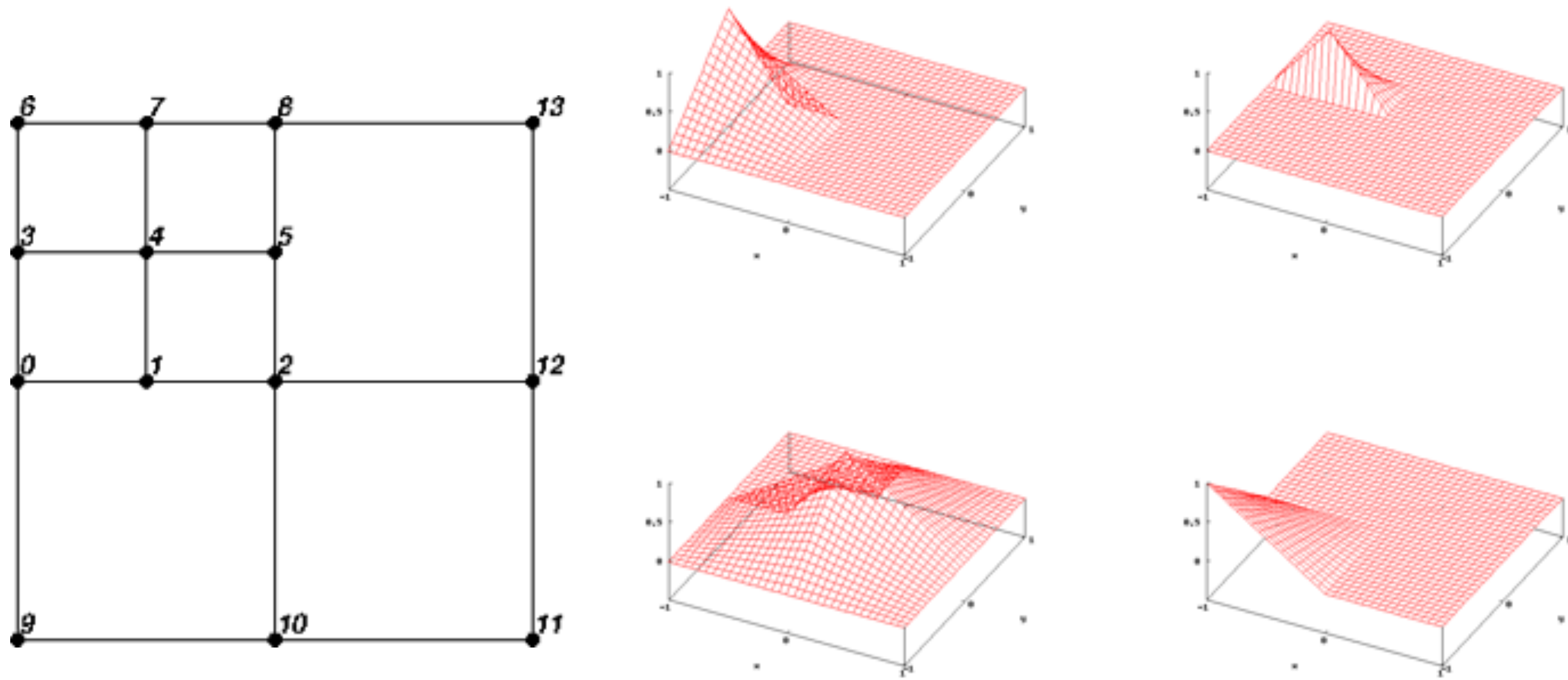
In other words:

$$V_h := \{v_h(x) = \sum_i V^i v_i(x) \text{ such that } v_h(x) \in C^0(\Omega)\}$$

MHPC
Master in High Performance Computing

Why constraints?

How do the shape functions look like:



Note: Not all of these functions are in V_h.

Which constraints?

Remember that we define V_h via constraints as:

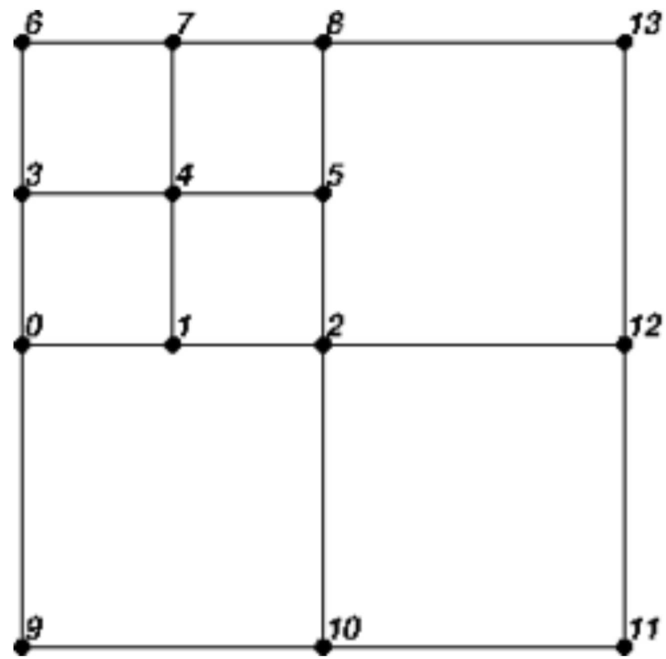
$$V_h := \{v_h(x) = \sum_i V^i v_i(x) \text{ such that } v_h(x) \in C^0(\Omega)\}$$

The only possible discontinuities are along edges 0–1–2 and 2–5–8.

The function is in fact continuous if it is continuous at vertices 1 and 5!

That is:

$$V_1 = \frac{1}{2}V_0 + \frac{1}{2}V_2, \qquad V_5 = \frac{1}{2}V_2 + \frac{1}{2}V_8$$



Which constraints?

As a general rule:

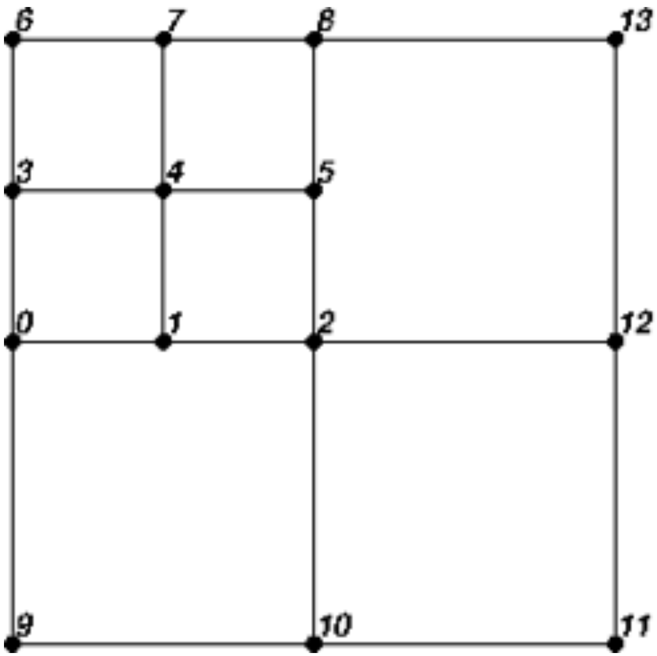
- When using hanging nodes, there is a subset I of *[0,n_dofs)* that is constrained
- These constraints have the form

$$V_i = \sum_{j=0, j\neq i}^{n-1} \alpha_{ij} V_j, \qquad \forall i \in I$$

where most of the alphas are zero

- Here, for example:

$$V_1 = \frac{1}{2} V_0 + \frac{1}{2} V_2, \qquad V_5 = \frac{1}{2} V_2 + \frac{1}{2} V_8$$



- We can write this as $CV = 0, \quad C \in \mathbb{R}^{\#\,\text{constraints} \times \#\,\text{dofs}}$

Representation in deal.II

In deal.II:

- The constraints *CV=0* for hanging nodes are represented by the deal.II class *ConstraintMatrix*
- *ConstraintMatrix* objects are built by the function *DoFTools::make_hanging_node_constraints*

Note: All of this works for *any* finite element, not just Q1. Furthermore, it also works for the *hp*-refinement case (see step-27).

Using constraints

Premise:

- The beauty of the FEM is that we do *exactly* the same thing on every cell
- Let us not destroy this property!
- That is: assembly on cells with hanging nodes should work exactly as on cells without.

Note: The mathematical and algorithmic details of dealing with constraints are complex (see Bangerth & Kayser-Herold, 2009). Therefore, let's discuss only the mechanics.

MHPC
Master in High Performance Computing

Using constraints

Define

$$\tilde{V}_h = \{ v_h(x) = \sum_i V_i \varphi_i(x) \}$$

$$V_h = \{ v_h(x) = \sum_i V_i \varphi_i(x) :\ v_h(x) \text{ is continuous in } \Omega \}$$
$$= \{ v_h(x) = \sum_i V_i \varphi_i(x) :\ CV = 0 \}$$

Approach 1 (step-6):

- Step 1: Build matrix/rhs A,f with all DoFs as if there were no hanging nodes.
- Step 2: Modify the matrices ("*condense*")
- Step 3: Solve $AU = F$
- Step 4: Get all components of *U* ("*distribute*")

Using constraints

Define

$$\tilde{V}_h = \{ v_h(x) = \sum_i V_i \varphi_i(x) \}$$

$$V_h = \{ v_h(x) = \sum_i V_i \varphi_i(x) : \ v_h(x) \text{ is continuous in } \Omega \}$$
$$= \{ v_h(x) = \sum_i V_i \varphi_i(x) : \ CV = 0 \}$$

Approach 2 (step-22):

- Step 1: Build *local* matrix/rhs with all DoFs as if there were no hanging nodes.
- Step 2: Modify when copying local contributions into global matrices ("*copy_local_to_global*")
- Step 3: Solve $AU = F$
- Step 4: Get all components of *U* ("*distribute*")