Finite element methods in scientific computing

Wolfgang Bangerth, Colorado State University

Lecture 15:

Adaptively refined meshes

Adaptive mesh refinement (AMR)

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

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

This suggests that 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)

But, recall from Lecture 3.9:

Commonly found features of solutions of PDEs:

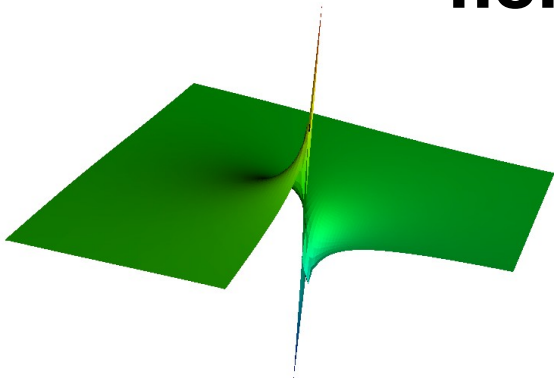
- "Smooth" in large parts of the domain
- Vary greatly in small parts of the domain
- May have kinks
- Singularities in corners

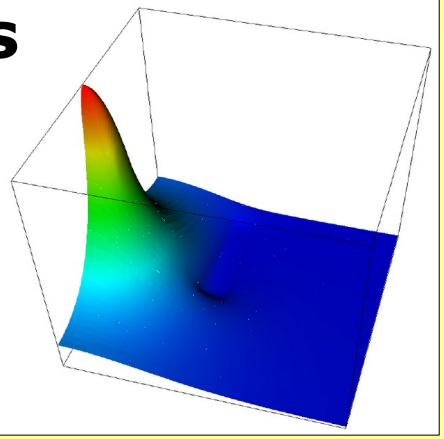
Adaptive mesh refinement (AMR)

But, recall from Lecture 3.9:

- Solutions are not C^∞, not even C^1
- Solutions can be shown to be in the "Sobolev space" H^1
- But functions in H^1 can be non-smooth *everywhere*
- In actual practice:

Solutions are *piecewise* smooth, with the exception of points or lower-dimensional sets of non-smoothness/singularities





Adaptive mesh refinement (AMR)

Indeed, a closer analysis shows:

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

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

In other words: To reduce the error, we *only* need to make the mesh fine *where the solution is not smooth*!

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

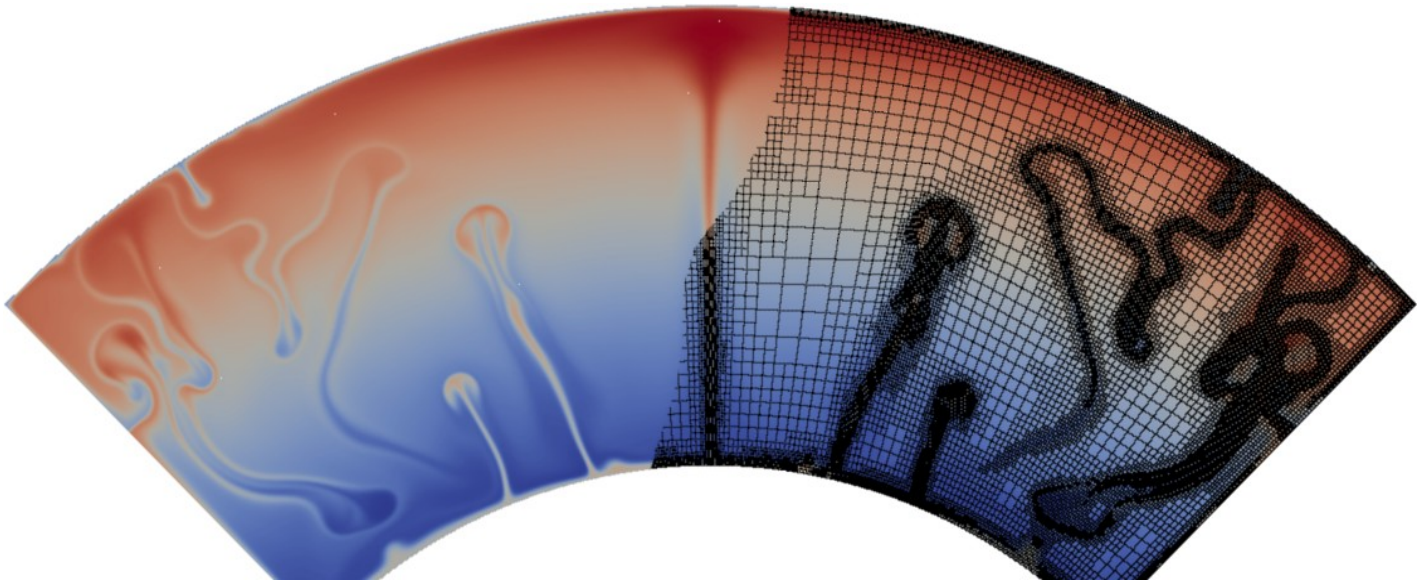
$$h_K \;\propto\; \frac{1}{\|u\|_{H^2(K)}}$$

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 up front 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:

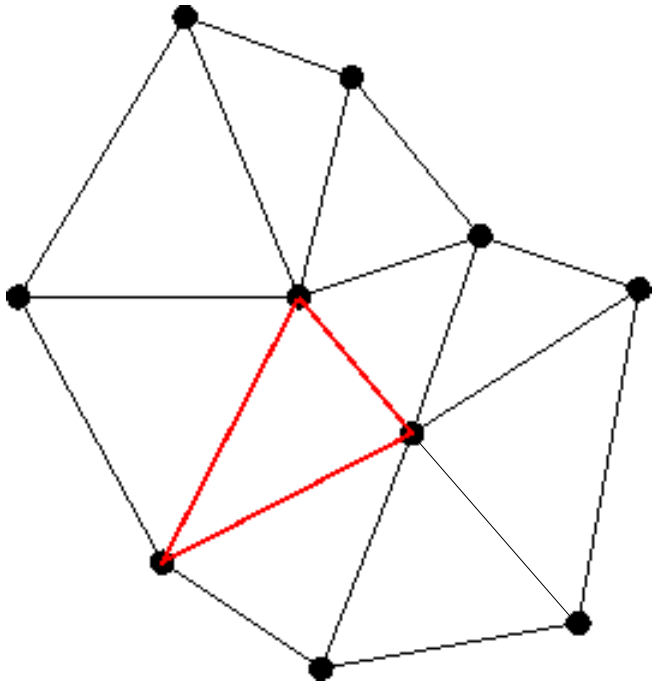
- *De novo* generation of a non-uniform mesh
- Longest edge refinement
- Red-green refinement

Note 1: Refining tetrahedra in 3d works analogously.

Note 2: There are also other strategies.

Longest edge refinement

Consider this mesh and a marked cell:

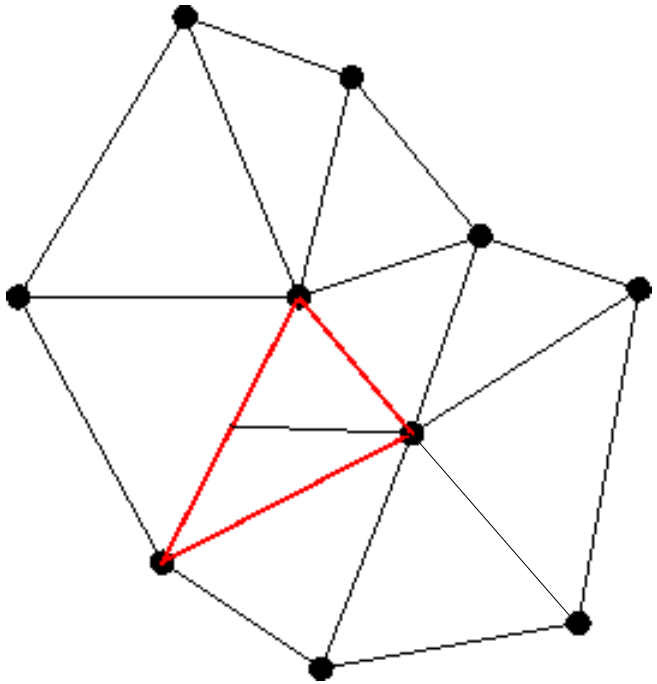


Algorithm:

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

Longest edge refinement

Consider this mesh and a marked cell:

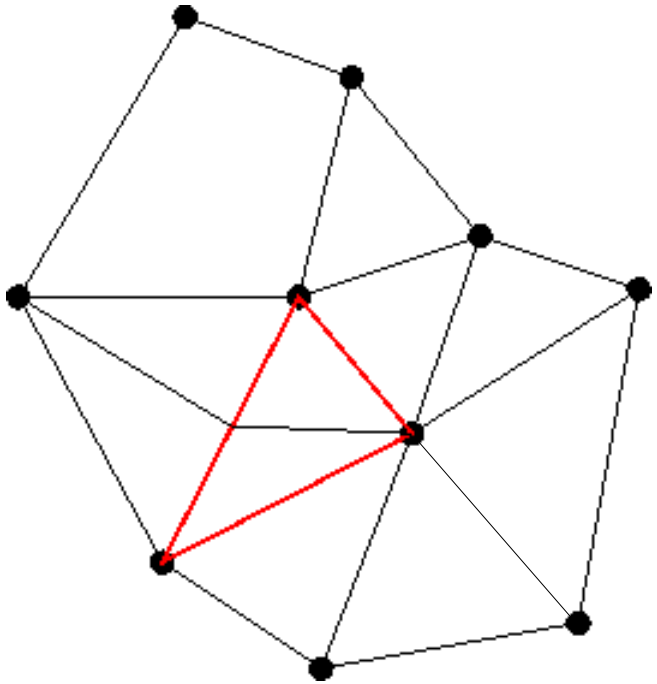


Algorithm:

- 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:

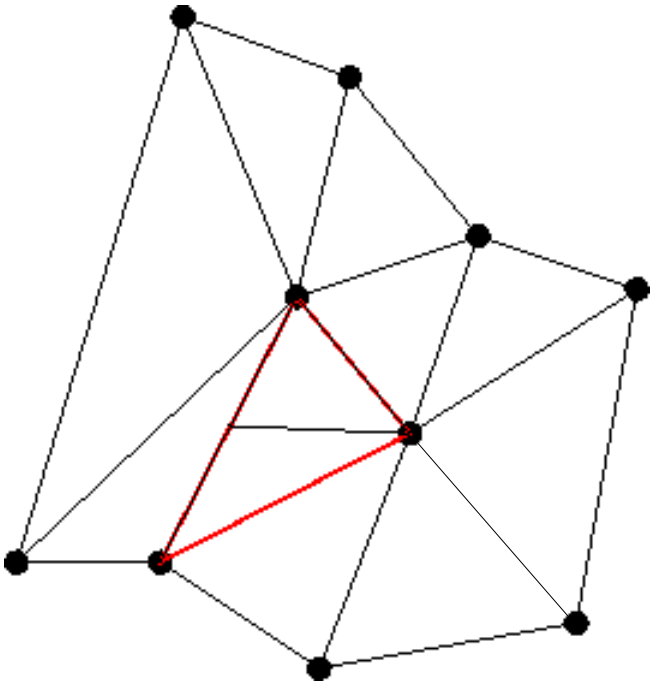


Algorithm:

- 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:

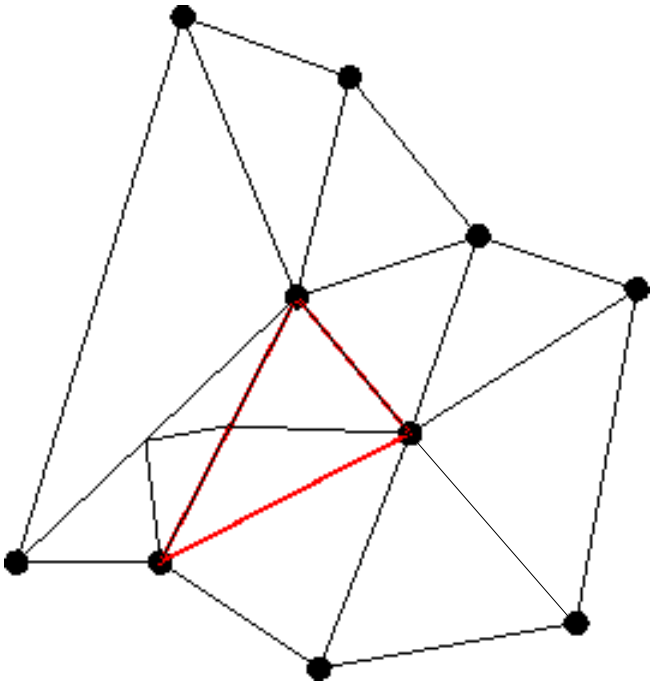


Algorithm:

- 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:

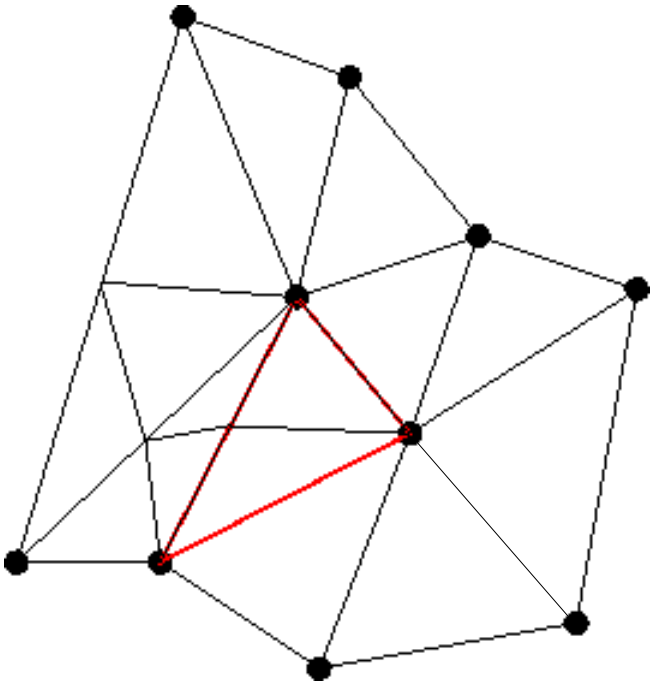


Algorithm:

- 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

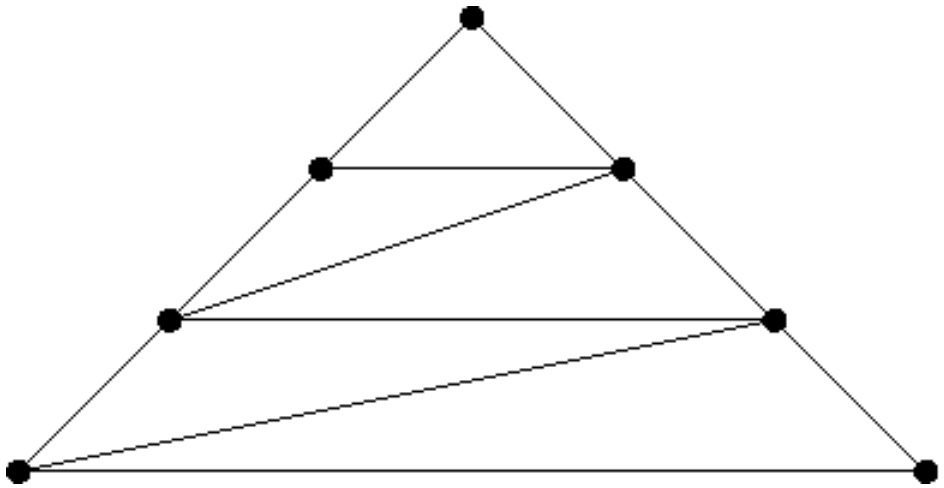
Longest edge refinement

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

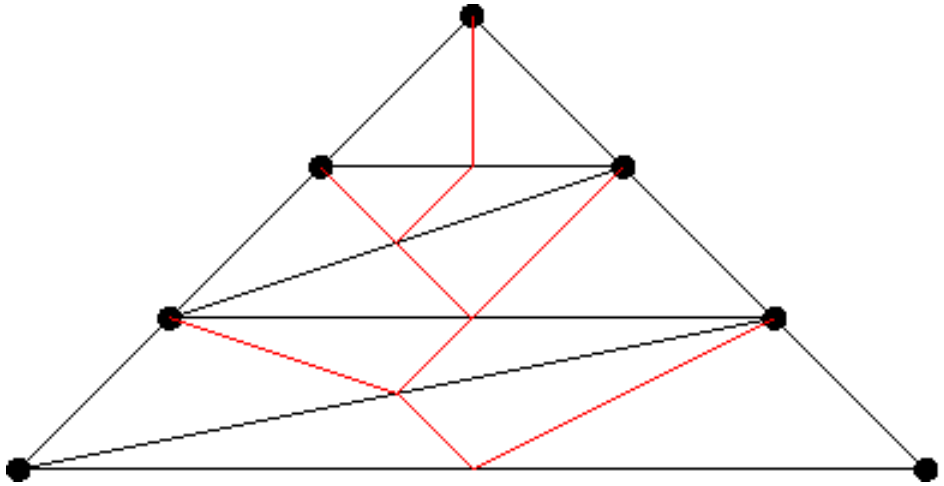
Example of runaway non-local refinement:



Goal: Refine the top-most cell.

Longest edge refinement

Example of runaway non-local refinement:

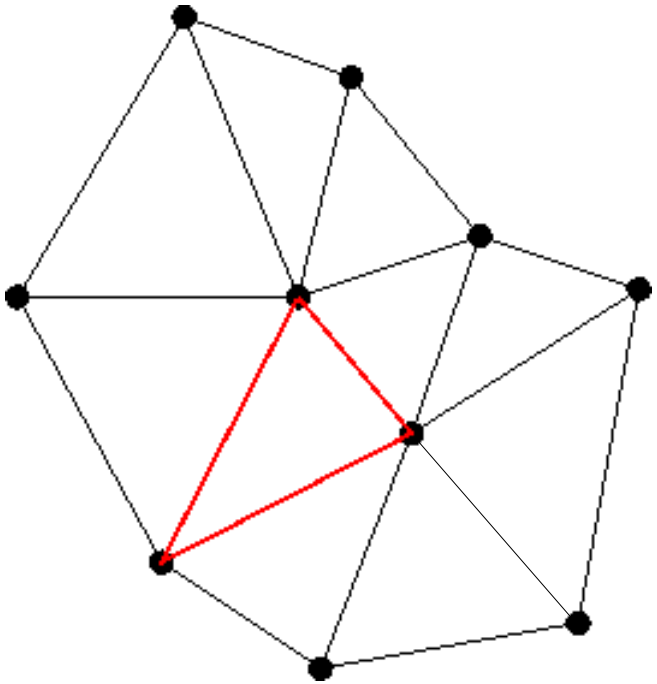


Result: *All* cells are refined!

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

Red green refinement

An alternative is as follows:

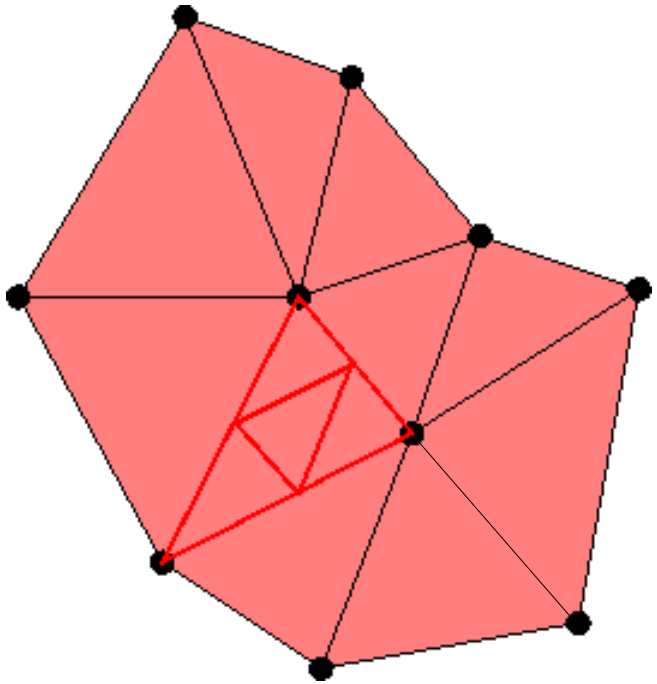


Algorithm:

– All cells start out as “red” cells

Red green refinement

An alternative is as follows:

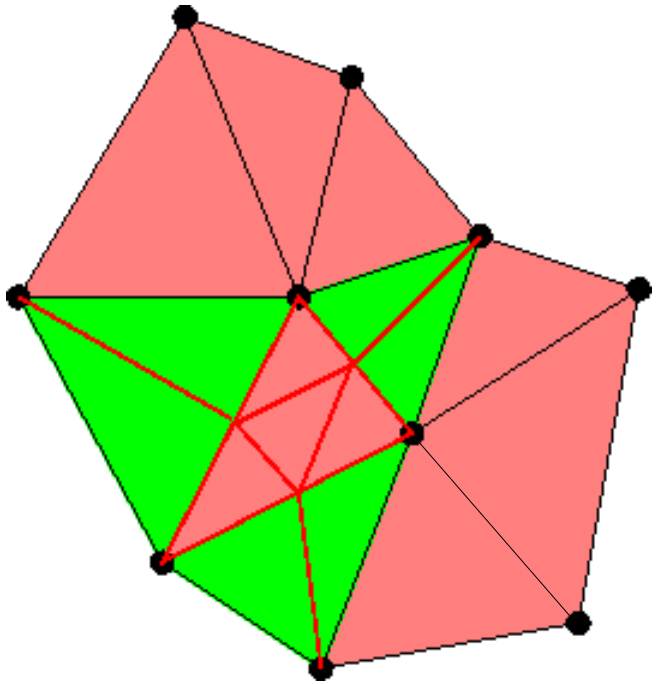


Algorithm:

- All cells start out as “red” cells
- The refinement of a “red” cell yields 4 “red” daughter cells

Red green refinement

An alternative is as follows:

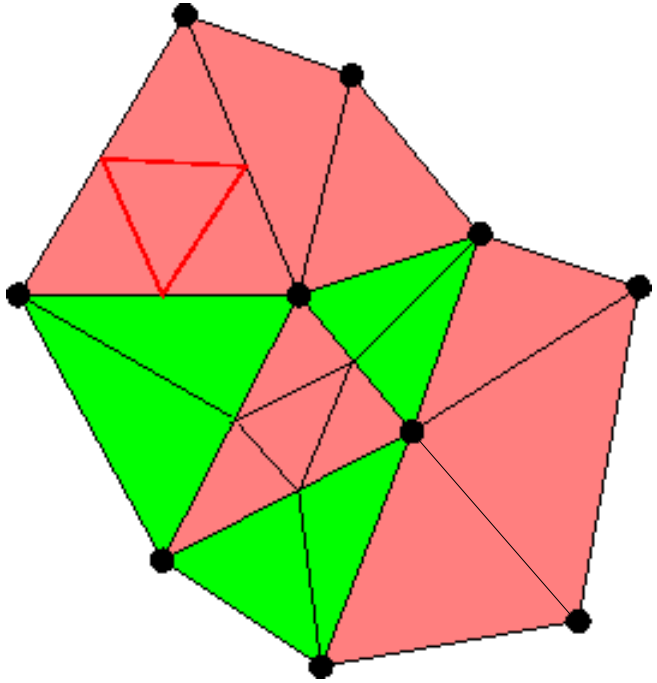


Algorithm:

- All cells start out as “red” cells
- The refinement of a “red” cell yields 4 “red” daughter cells
- Cells with “hanging nodes” are halved and become “green”

Red green refinement

"Green" cells are second-class citizens upon further refinement:

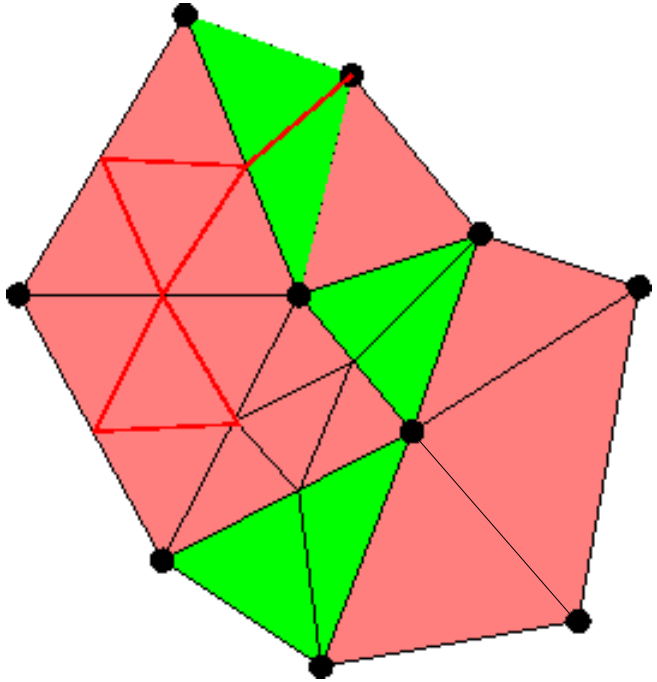


Algorithm:

– Marked "red" cells are refined as always into "red" cells

Red green refinement

"Green" cells are second-class citizens upon further refinement:



Algorithm:

- Marked "red" cells are refined as always into "red" cells
- "Green" cells are first undone and then refined as "red" cells

Red green refinement

Analysis:

Pros:

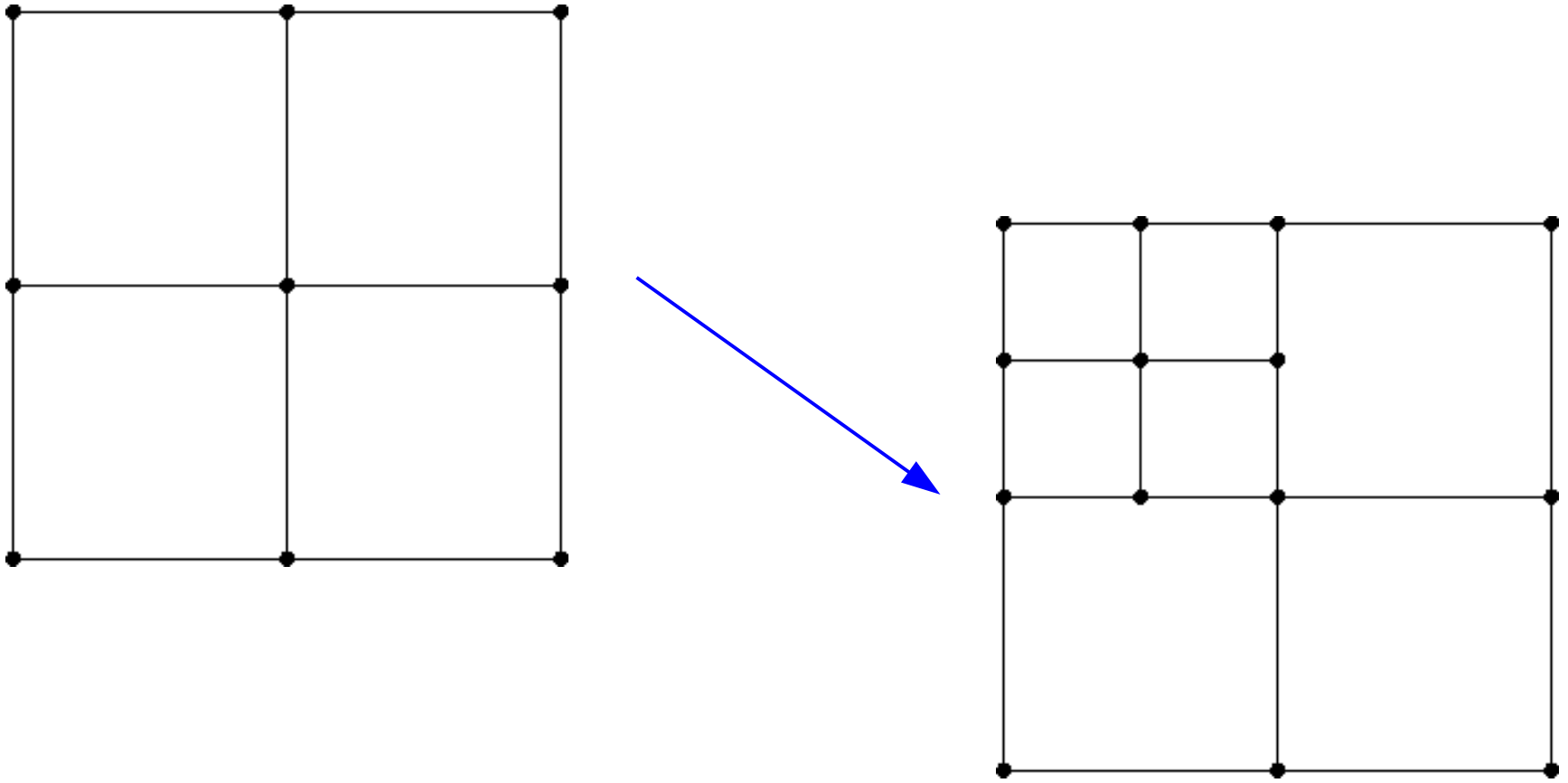
- The algorithm is designed to keep the triangles from degenerating
- Children of "red" cells are congruent to their mother
- No run-away refinement

Cons:

- Triangles can degenerate by a factor of 2 when converted to "green" cells
- More implementation effort required

Quad/hex refinement

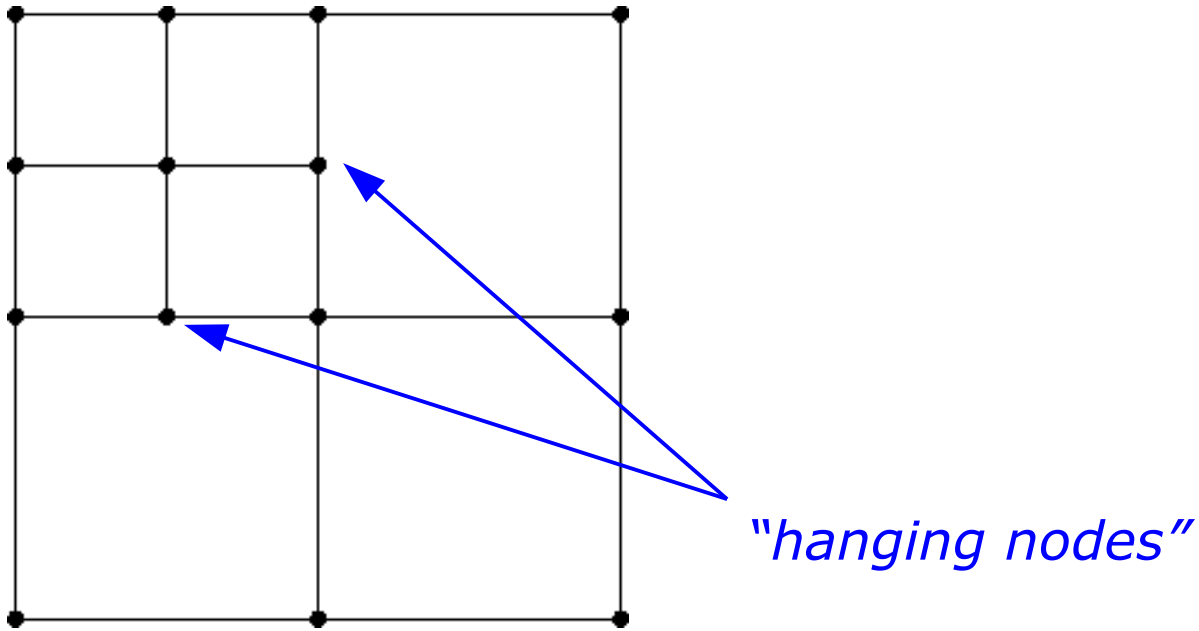
For quads/hexes, the situation is more difficult:



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!

Finite element methods in scientific computing

Wolfgang Bangerth, Colorado State University