Numerical solution of PDEs using the Finite Element Method

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

19-23 March 2018

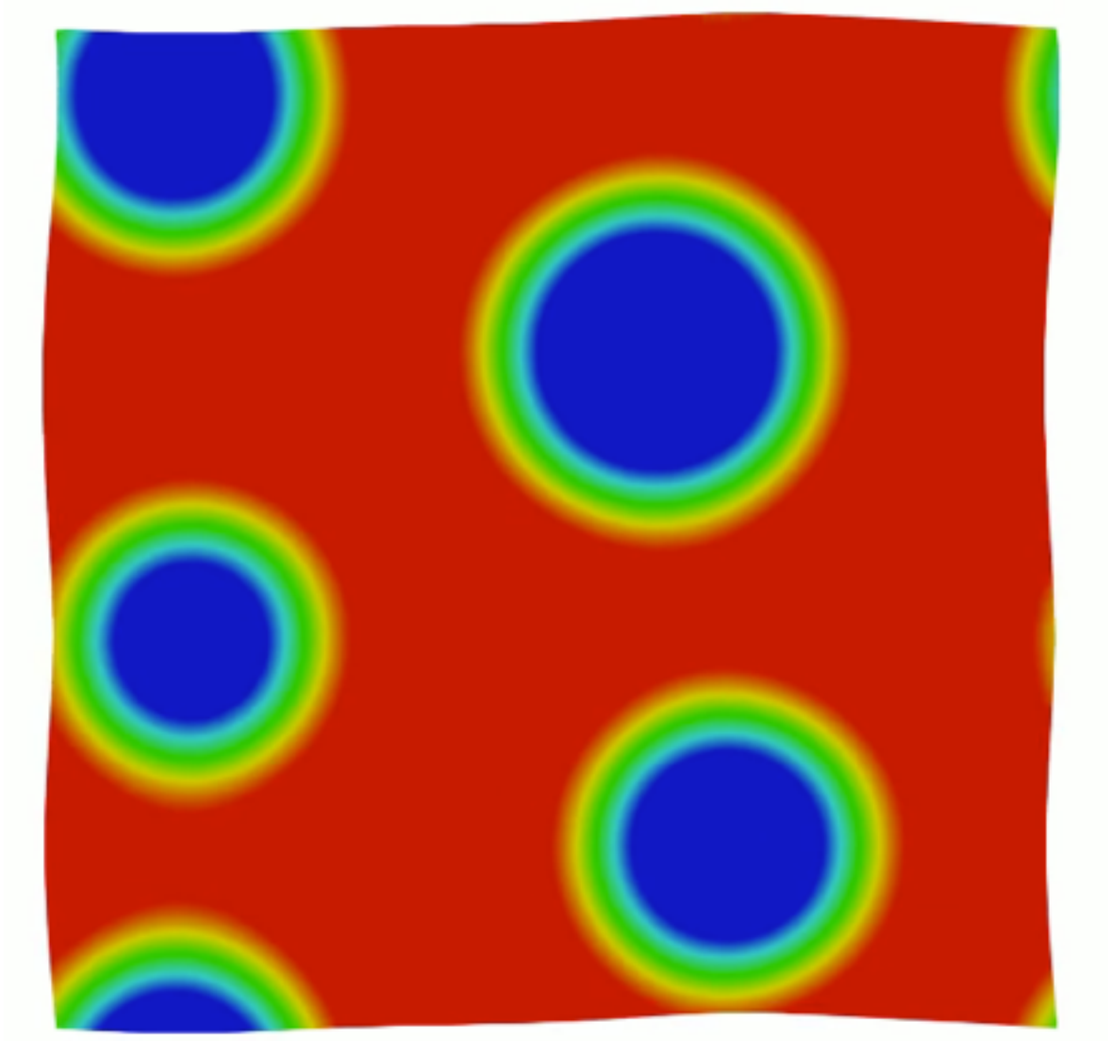
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Course goals

- Learn the fundamentals of deal.II
  - Commonly used data structures, their interface
  - Structure of finite element problems
  - Good implementation practices
  - Navigate the documentation

Course goals

- Our goals (challenge) for you:
  Implement a solution to the uncoupled Cahn-Hilliard equation
  - Scalar problem
    - Nonlinear, time-dependent
    - Adaptive mesh refinement
    - Parallelised assembly and sparse linear solver
    - Verified assembly of linear system
    - Submit this as a tutorial to deal.II



Course schedule

Time	Duration	Content	Speaker	Content	Speaker
		MONDAY 19.03.2018		**TUESDAY 20.03.2018**	
09:30	1.25 hours	Introduction First steps	JPP	Local refinement Hanging nodes	JPP
		COFFEE / TEA			
11:15	1.25 hours	Introduction to FEM	LH	Local (adaptive) refinement Computing errors	LH
		LUNCH			
14:00	1.25 hours	Solving Poisson's equation	JPP	Shared memory parallelisation	JPP
		COFFEE / TEA			
15:45	1.25 hours	Exercises, Q&A	JPP, LH	Exercises, Q&A	JPP, LH
		WEDNESDAY 21.03.2018		**THURSDAY 22.03.2018**	
09:30	1.25 hours	MPI parallelisation: Part 1	JPP	Useful utility classes Git workflow	JPP
		COFFEE / TEA			
11:15	1.25 hours	Exercises, Q&A	JPP, LH	Time dependent problems Solution transfer	JPP
		LUNCH			
14:00	1.25 hours	MPI parallelisation: Part 2	LH	Automatic differentiation	JPP
		COFFEE / TEA			
15:45	1.25 hours	Exercises, Q&A	JPP, LH	Exercises, Q&A	JPP, LH

FRIDAY 23.03.2018

Project

How the course will be run

- Each module will have a lecture
    - Present salient information
    - Put what we'll learn into context
- Then we'll walk through aspects of the tutorials together
    - Discuss important functionality
        - What it does
        - How it works
        - Caveats and tips
- Remainder of the lecture will be spent doing some exercises
    - Suggestion: Work in groups of two/three
    - Continued at in the last session of the day

Resources

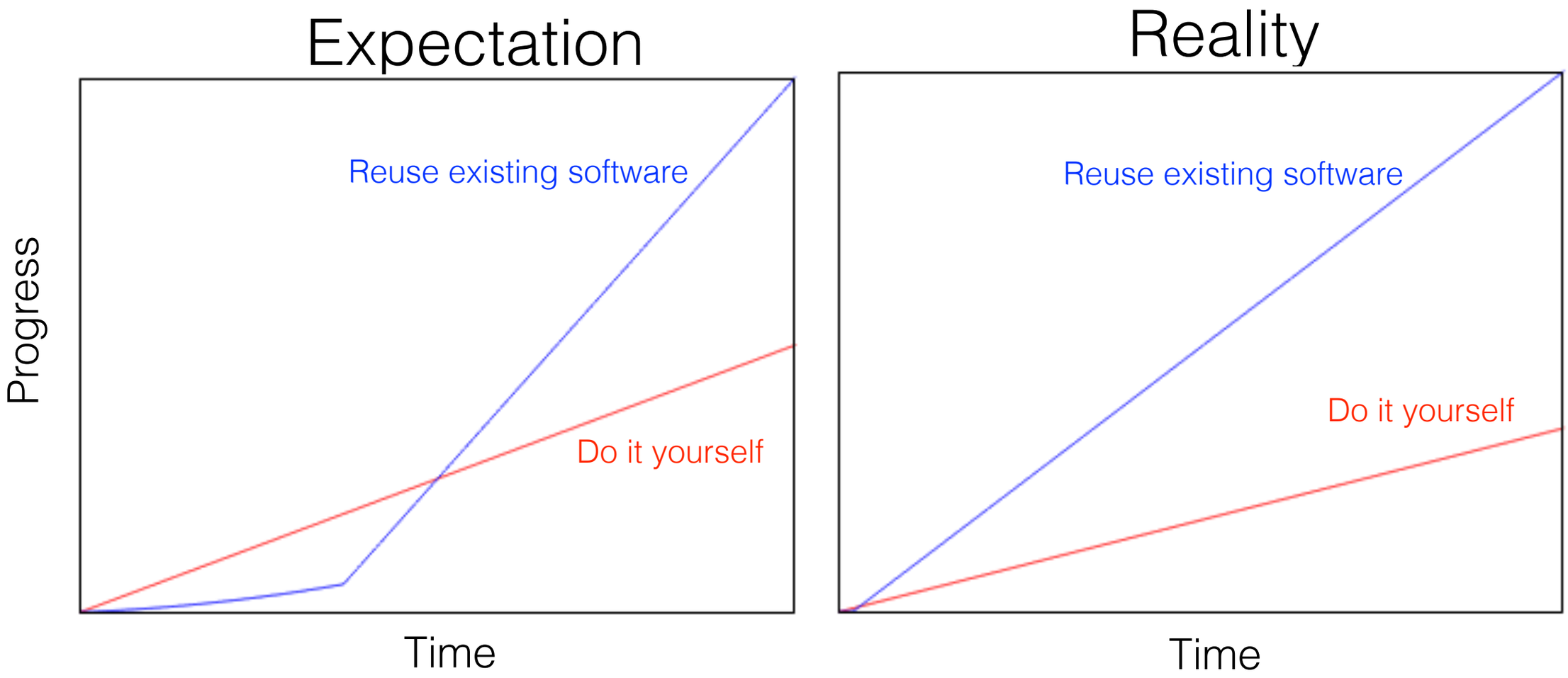
- deal.II user manual
  - https://www.dealii.org/developer/doxygen/deal.II/index.html
  - https://www.dealii.org/developer/doxygen/deal.II/modules.html
  - https://www.dealii.org/developer/doxygen/deal.II/DEALGlossary.html
- deal.II tutorials and code gallery
  - https://www.dealii.org/developer/doxygen/deal.II/Tutorial.html
  - https://www.dealii.org/developer/doxygen/deal.II/CodeGallery.html
- Us :-)
  - Don’t hesitate to ask questions

Introduction to deal.II

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

19 March 2018

Why use deal.II (or any other PDE toolbox)?

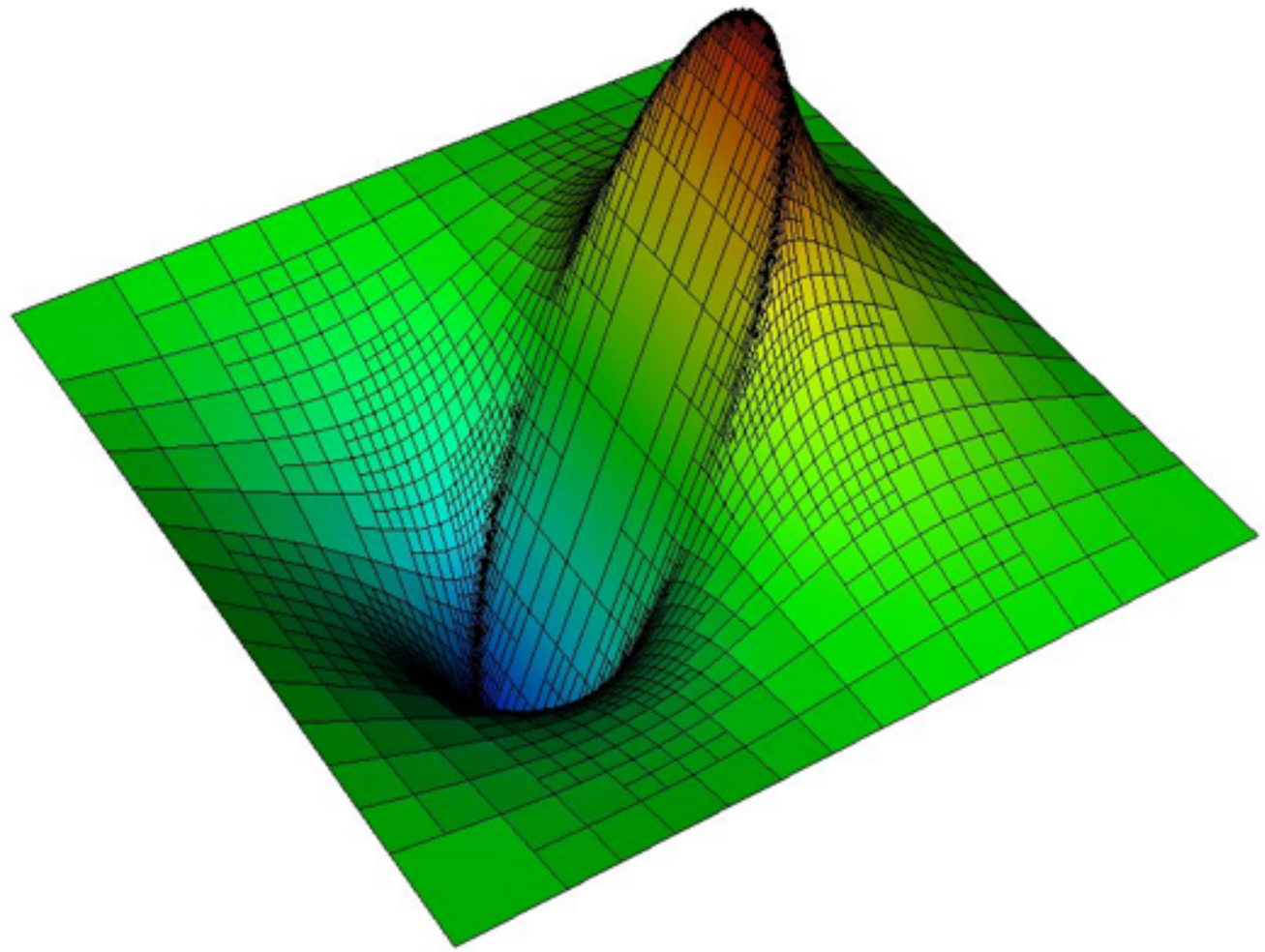


- Applies to:
  - Users
  - Developers
- “The secret to good scientific software is (re)using existing libraries”

What is deal.II?

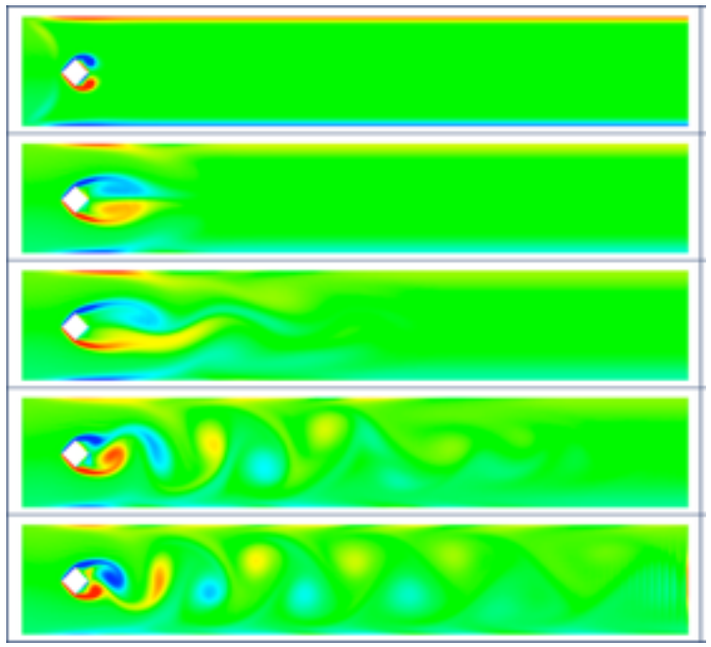
Differential Equation Analysis Library

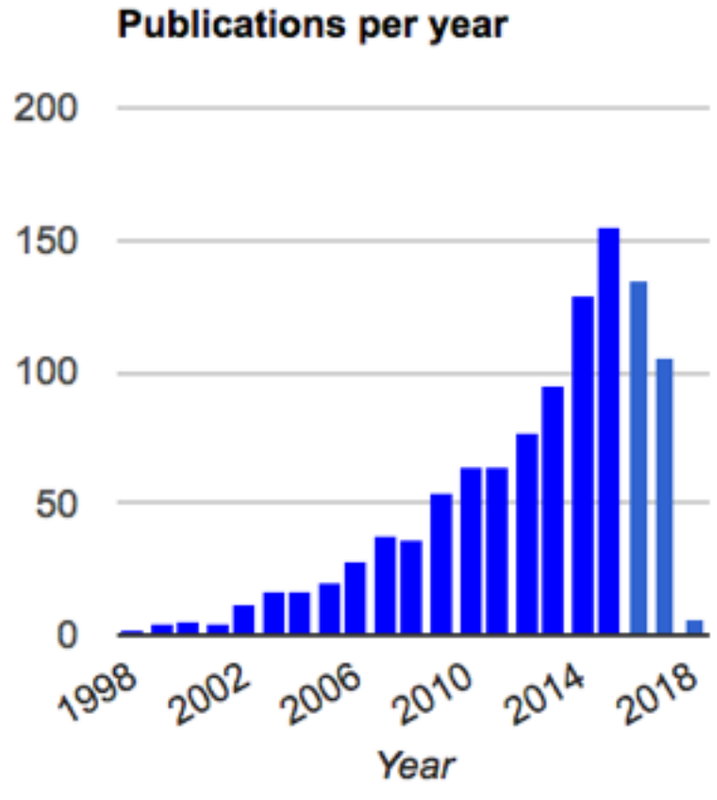
- Flexible open-source finite element toolkit
    - All the support functionality required to describe and solve a FE problem (PDEs)
    - Optimised for speed
    - Heavily tested
        - Many error checks (debug mode)
        - ~9500 regression tests run continuously
        - https://cdash.kyomu.43-1.org/index.php?project=deal.II
    - Part of SPEC CPU 2017 benchmark
- Templated C++ library (Object Orientated)
    - Dimension independent programming
- Portable
    - OS, architecture, compiler
- Origins
    - Study mesh adaptivity and error estimation
    - Now used in many other applications, frameworks



What is deal.II?

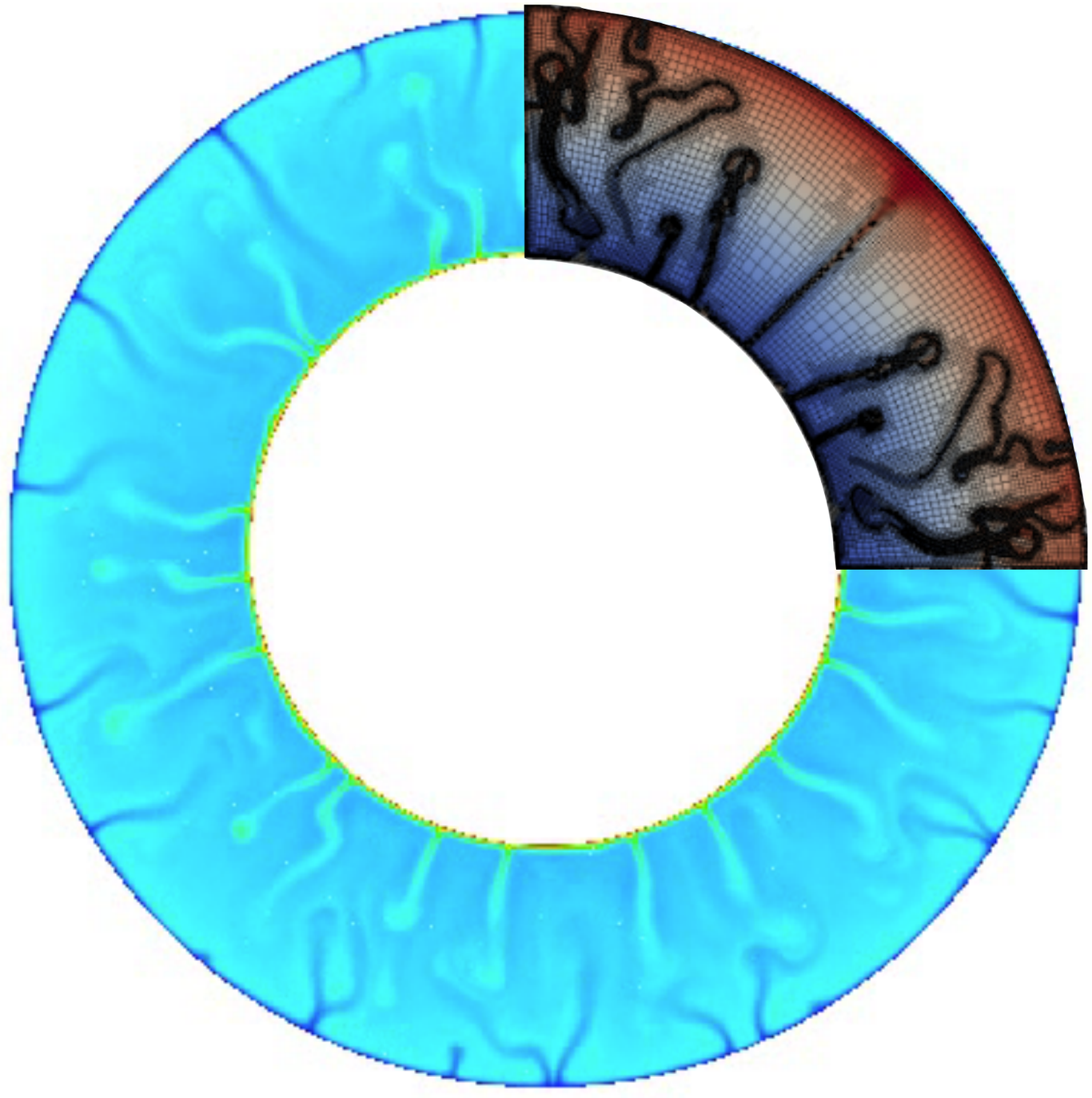
- Heavily documented
  - Over 10000 pages of interface documentation
  - Numerous tutorials
    - Illustrate functionality
    - Present methods to solve problems
- Quite widely used, and growing
- Active community
  - Approachable developers
  - Helpful online forum





Classes of problems solved using deal.II

- Geomechanics
- Fluid and gas dynamics
- Porous media
- Fluid-structure interaction
- Boundary element method
- Topology optimisation
- Medical image reconstruction
- Structural mechanics
- Biomechanics
- Crystal growth
- Gradient and crystal plasticity
- Contact mechanics
- Continuum-atomistic coupling
- Magneto- and electro-elasticity
- Thermo-plasticity

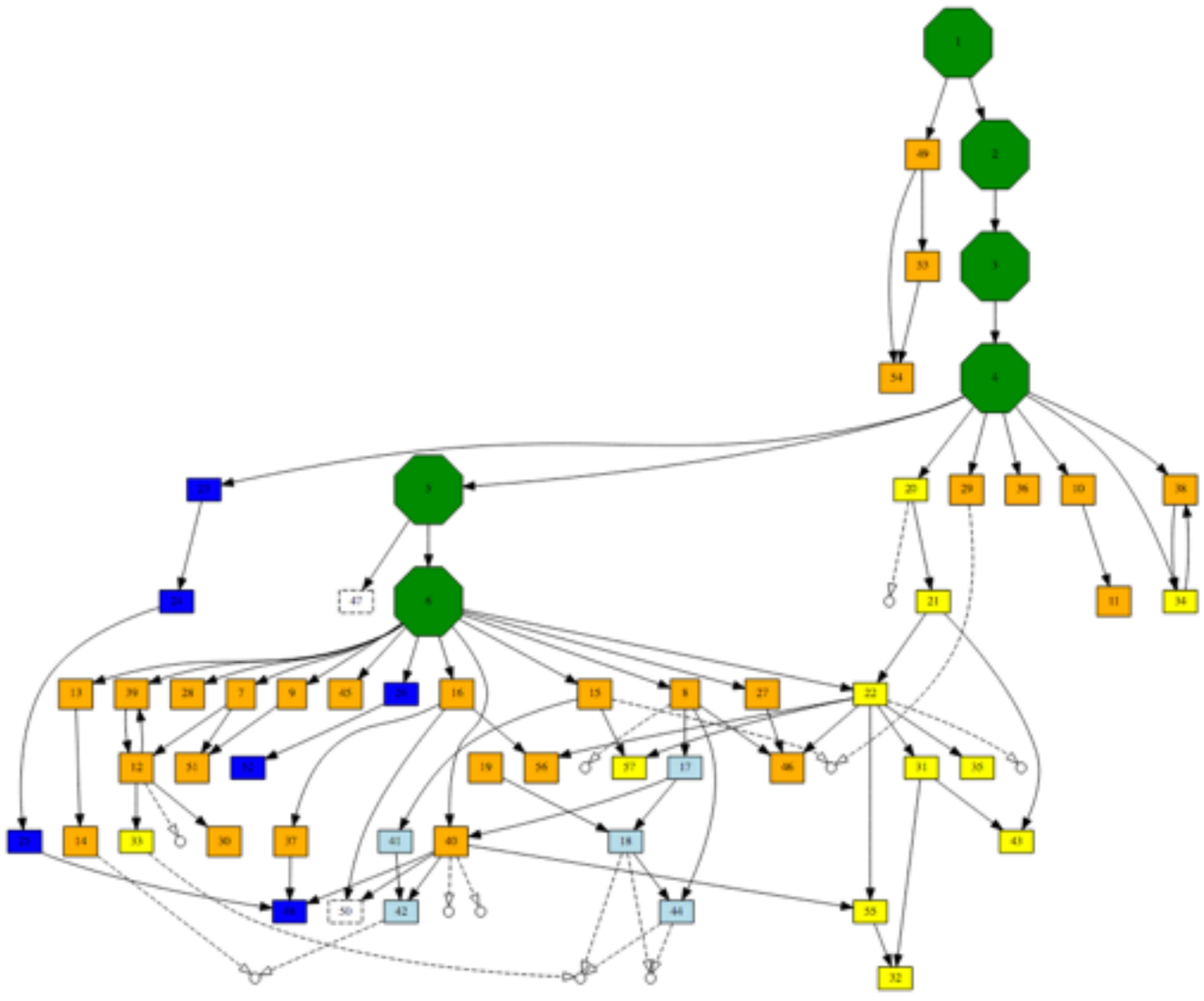


What deal.II is not

- A black box
  - Can’t throw any problem at it
  - Won’t do anything more than you ask it to
- Knows little* about
  - Numerical methods
  - Problem-specific implementation
    - Finite-element perspective
    - Physical conditions

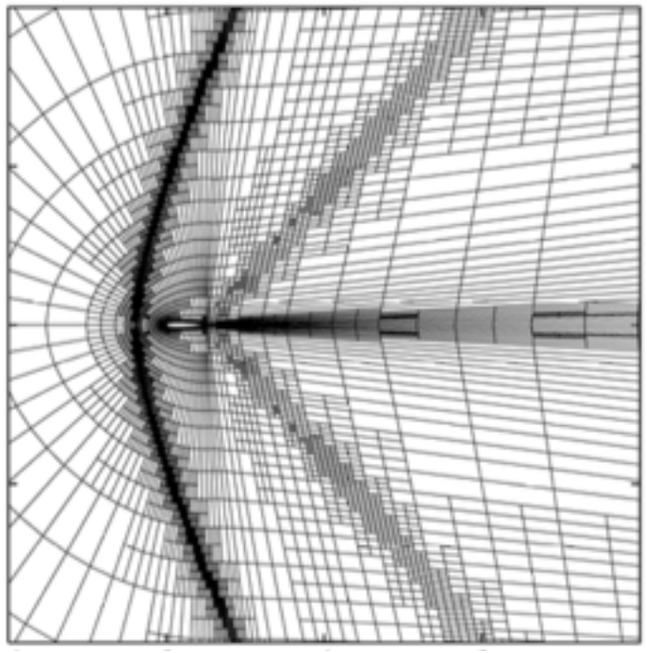
How deal.II will help you

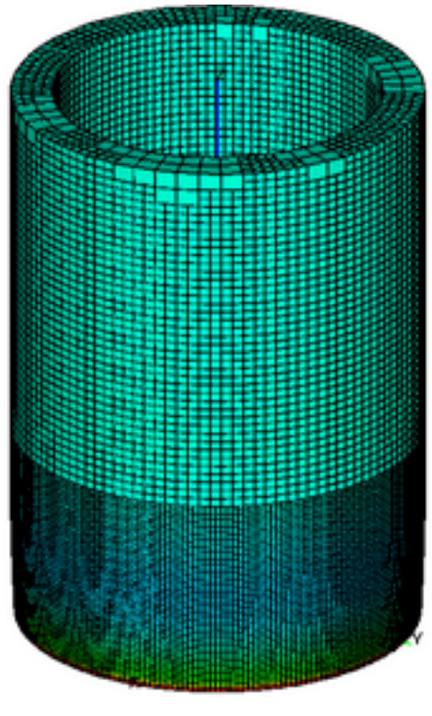
- Unified and well thought out data structure
  - Problem implementation
- Many tutorials
  - Baseline from which to build on
  - Demonstrate how to use features
- Comprehensive debugging support
  - Error messages everywhere!
- Some built in numerical tools
- Integration with advanced frameworks
  - Nonlinear solvers
  - Time integrators

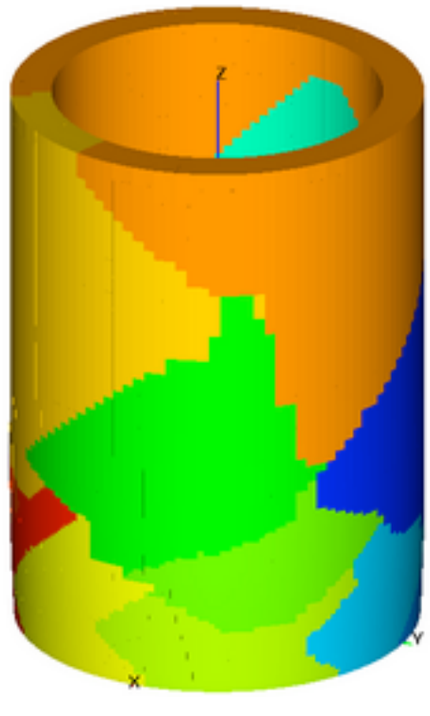


Fundamental capabilities and frameworks

- Mesh adaptivity
- Full and sparse linear algebra
  - Built in tensor, dense matrix/vector classes
  - BLAS and LAPACK integration; GSL
  - Built in linear solvers and preconditioners
  - Eigenvalue solvers
- Parallelisation
  - MPI
    - Linear algebra libraries (PETSc, Trilinos)
    - Distributed meshes ⟶ Billion DoF problems
  - Threading
  - Vectorised numbers (AVX extensions)
- Pre/post-processing

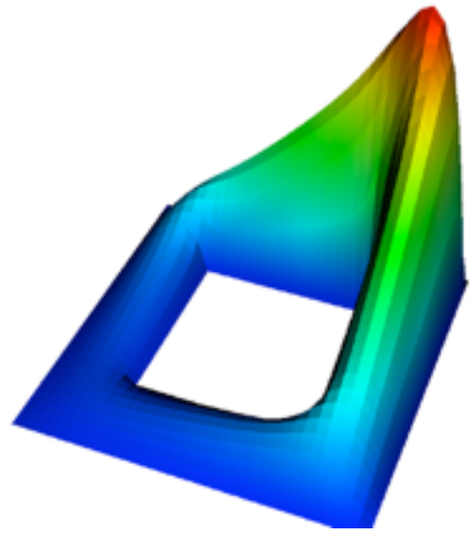


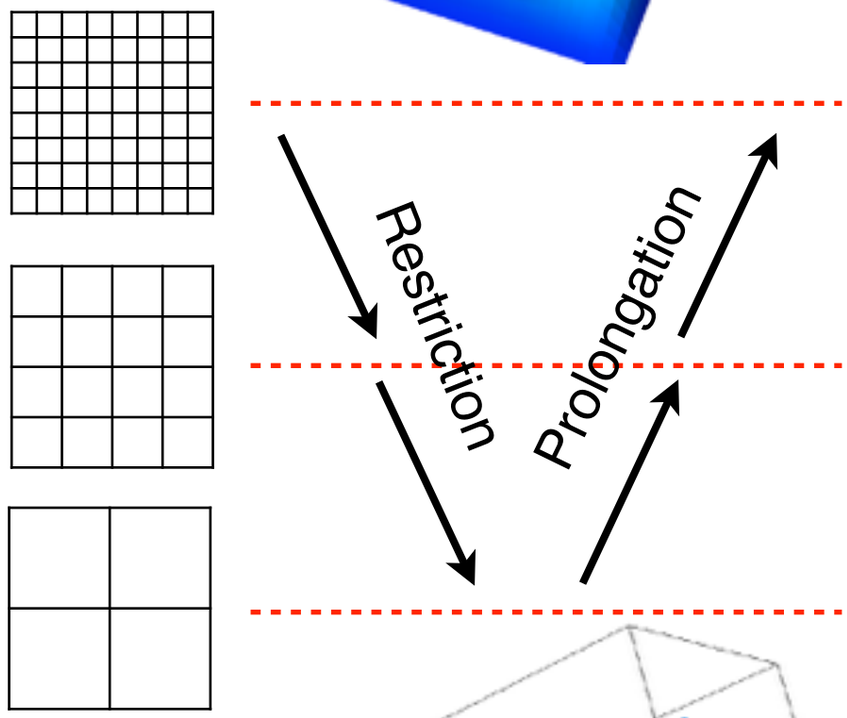


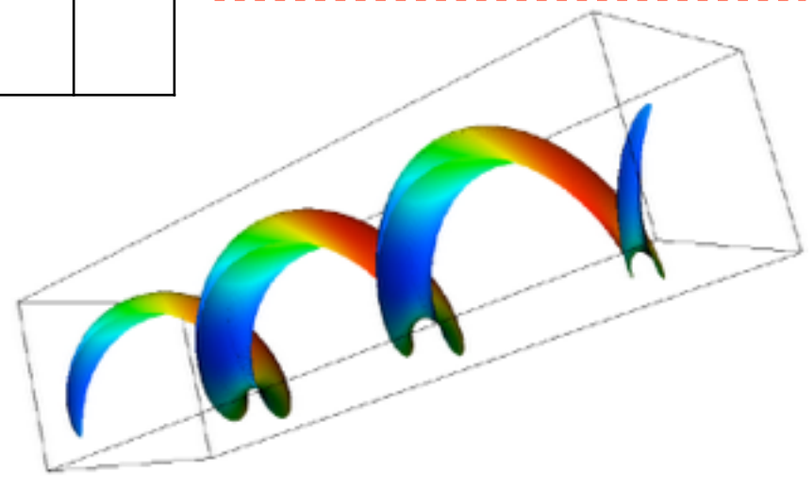


Advanced capabilities and frameworks

- hp-finite element support
- Meshworker
  - Assembly assistance
  - Functions to perform assembly for specific problem classes
- Geometric multi-grid
  - Using coarse grid as preconditioner to solution for finer grid
- Matrix-free
  - No explicit storing of matrix elements
  - Huge time-savings possible
- Charts and manifolds
  - Accurate description of topologically complex objects







How deal.II is developed

- Open repository on GitHub
  - https://github.com/dealii/dealii
- Anyone can contribute!
- We encourage all to participate

Getting started with deal.II

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

19 March 2018

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Aims for this module

- Gain familiarity with two core classes
  - Triangulation
  - DoFHandler
- Create and interrogate meshes
- Create and interrogate sparsity patterns

Reference material

- Main page
  https://dealii.org/8.5.1/doxygen/deal.II/index.html

- Tutorials

  - Step-1
    https://dealii.org/8.5.1/doxygen/deal.II/step_1.html

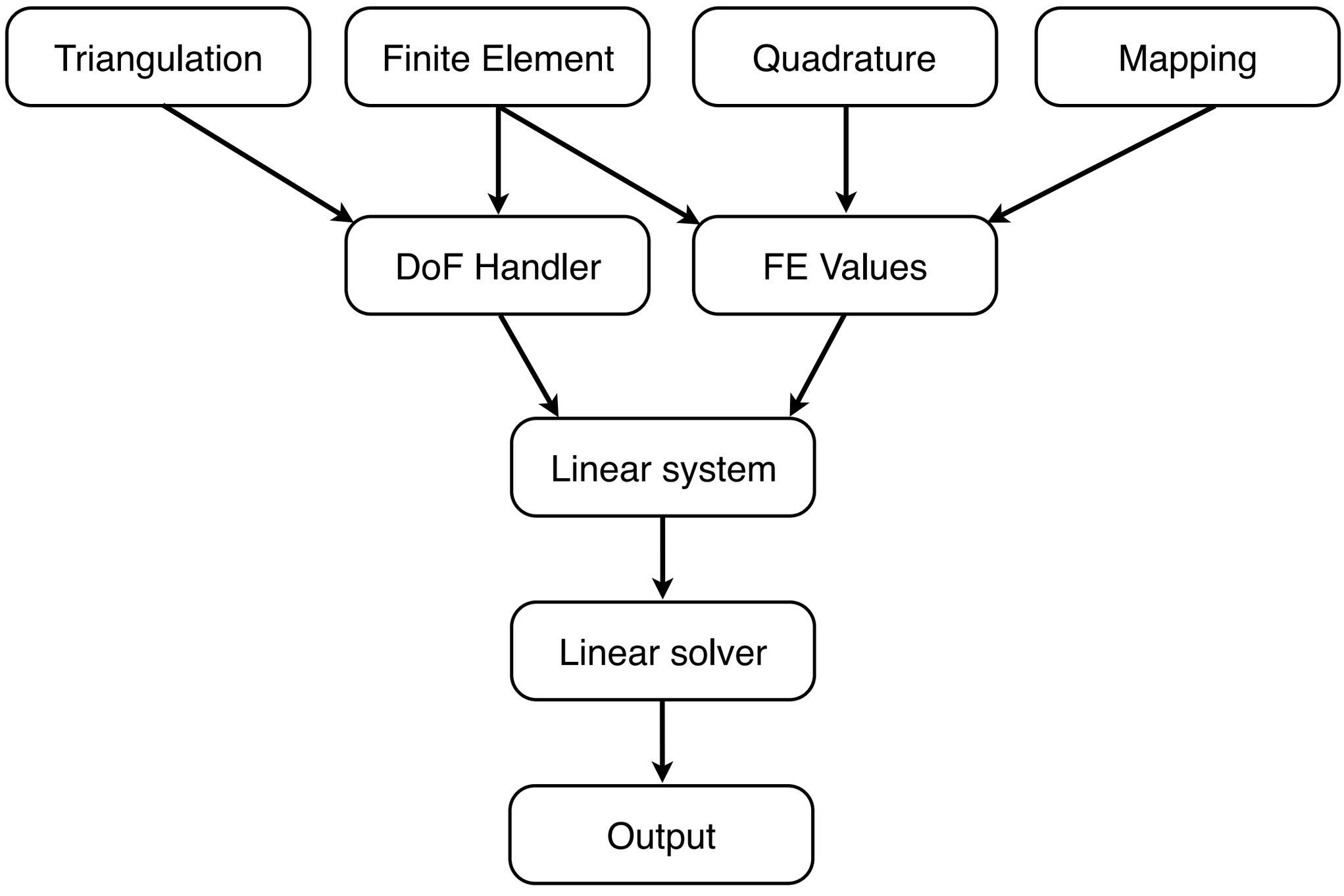
  - Step-49
    https://dealii.org/8.5.1/doxygen/deal.II/step_49.html

  - Step-2
    https://dealii.org/8.5.1/doxygen/deal.II/step_2.html

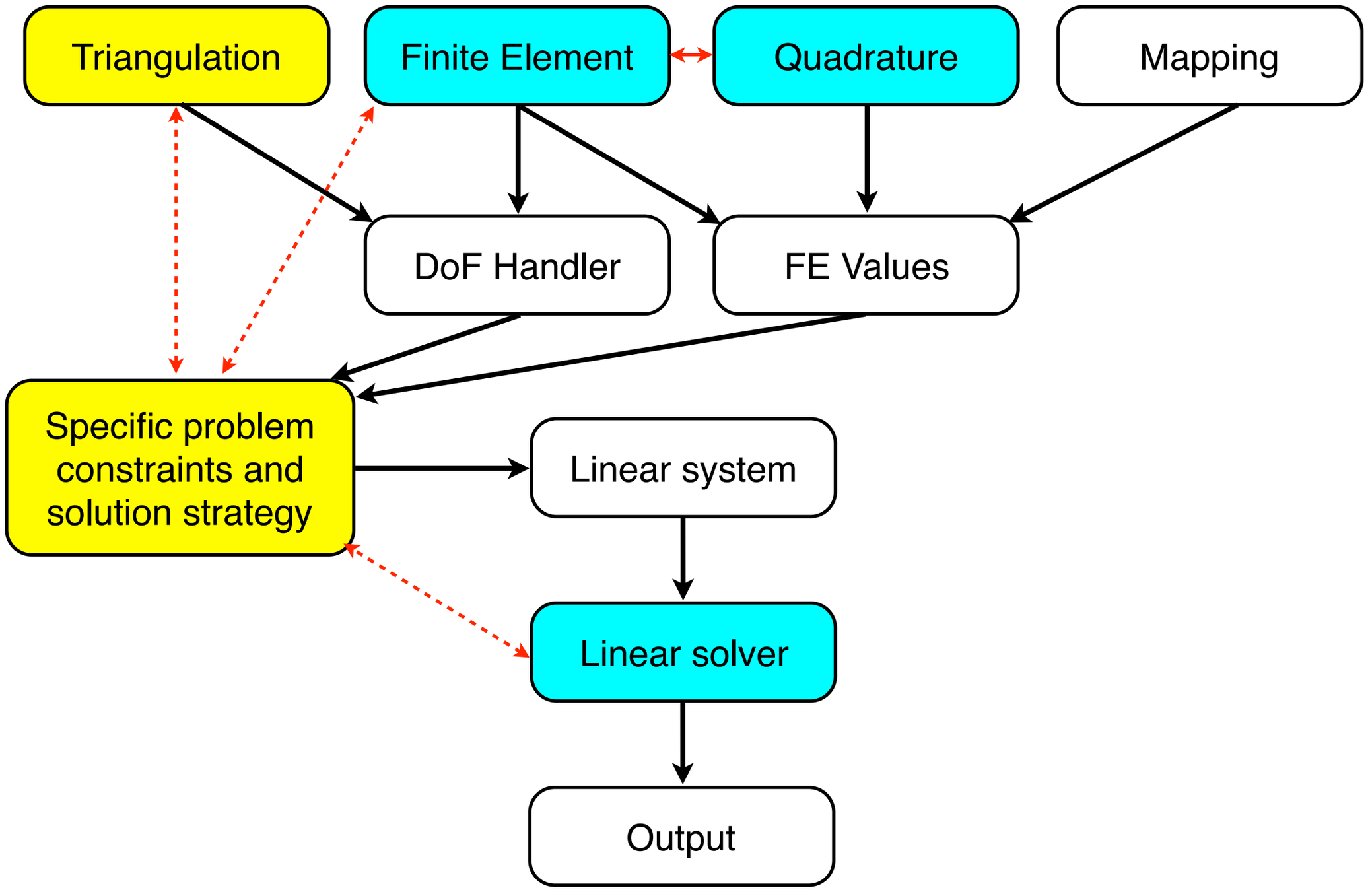
First and biggest tip

- Program defensively
  - Program and test in debug mode
    - Additional compiler warnings
    - Add assertions
  - Perform studies in release mode

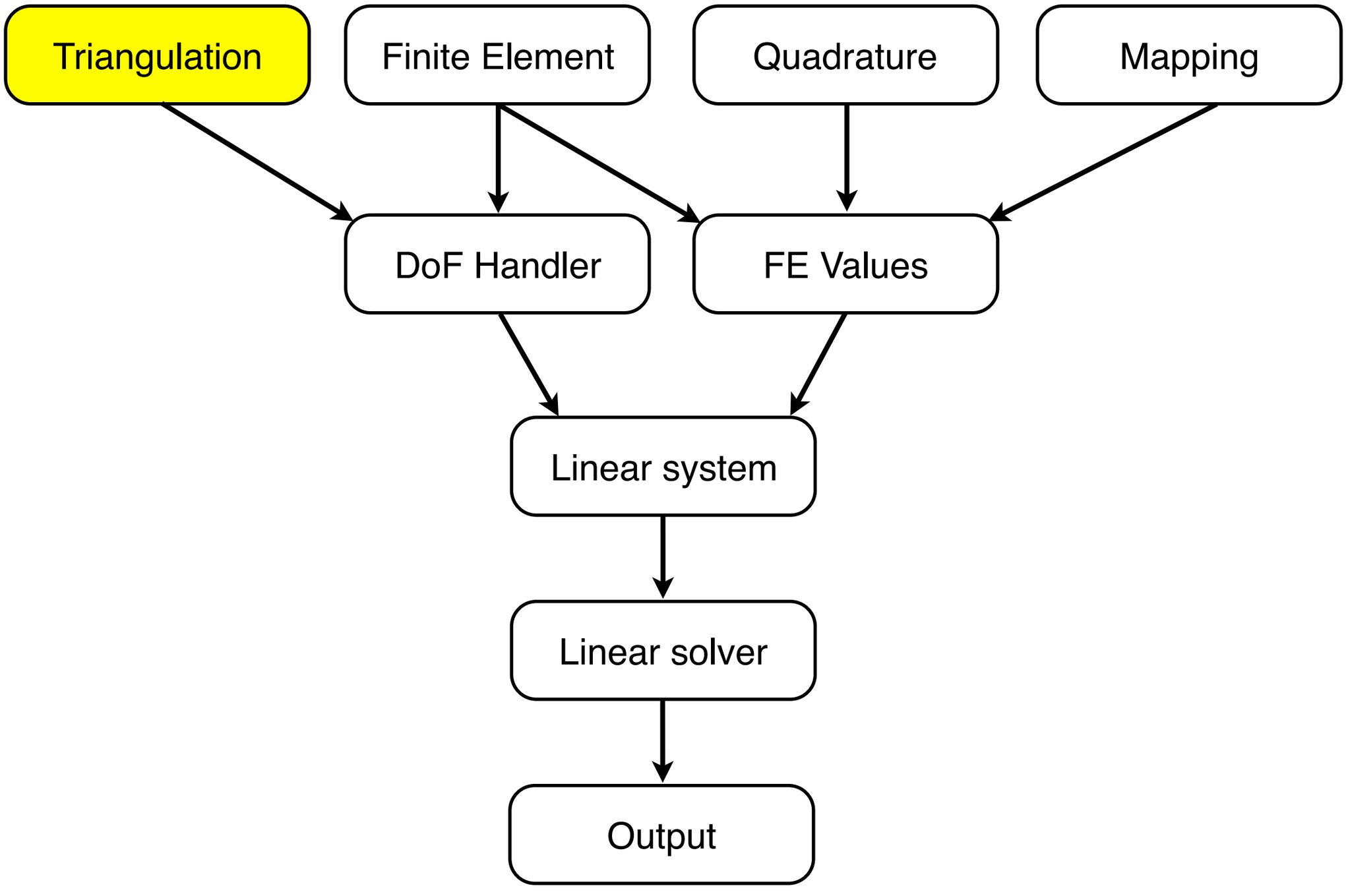
Structure of a prototypical FE problem



Structure of a prototypical FE problem

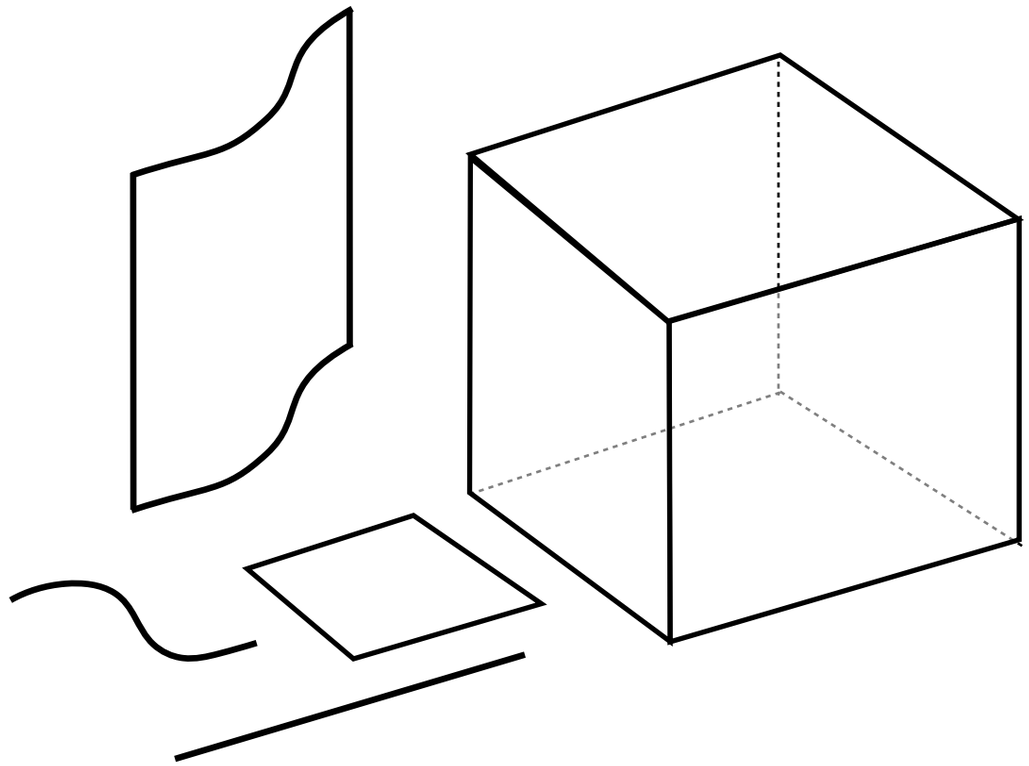


Structure of a prototypical FE problem



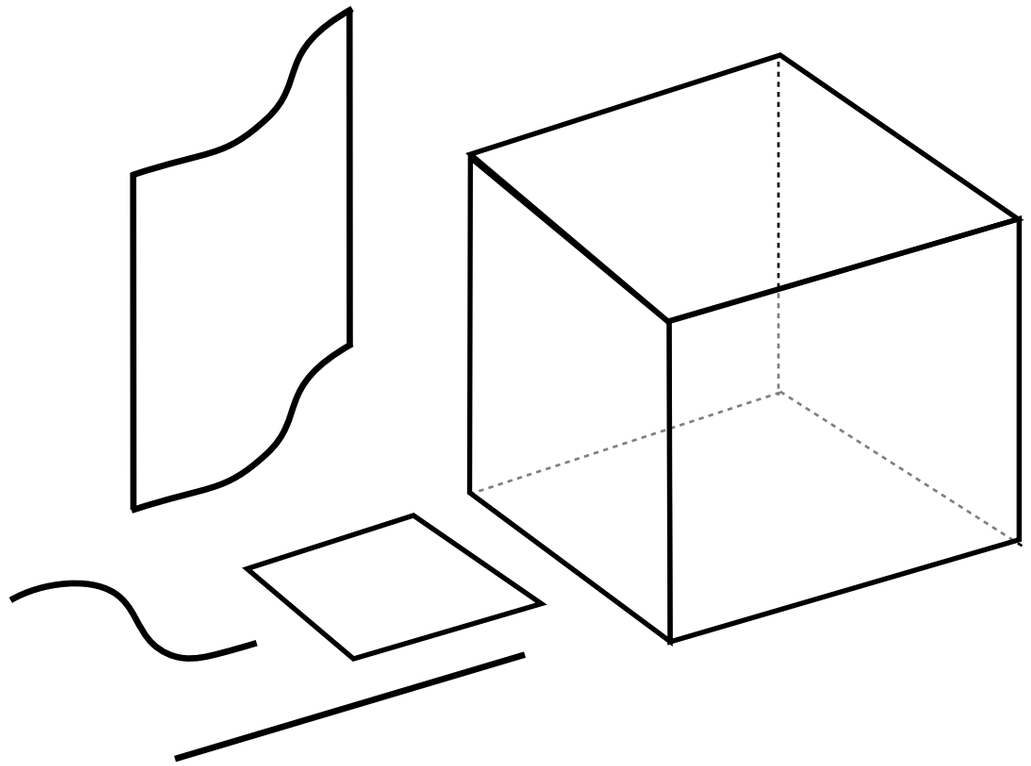
Interaction with geometry: the Triangulation class

- Describes problem geometry
  - Support for lines, quad, hex elements
  - Conceptually even higher order!
  - Structured/unstructured meshes
    - Co-dimension 1 or 2 case
- Grid creation
  - Built-in basic grid generation and manipulation tools
  - Can read in grids



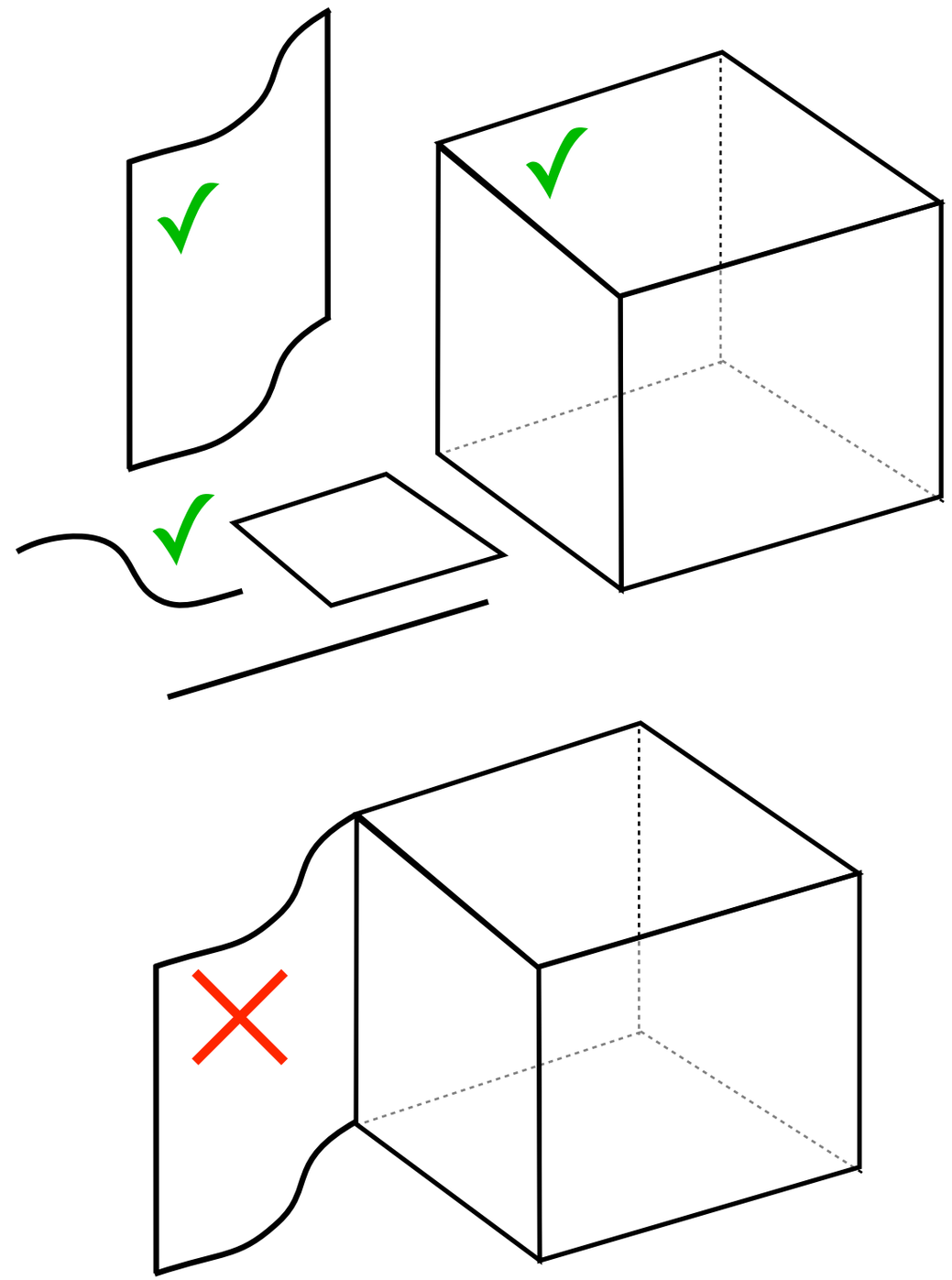
Interaction with geometry: the Triangulation class

- Assign helper ID's
  - Materials
  - Boundaries
  - Manifolds
- Allows storage of custom data-structure attached to each cell/face
- Cells know about neighbour cells
  - Useful for DG methods



Interaction with geometry: the Triangulation class

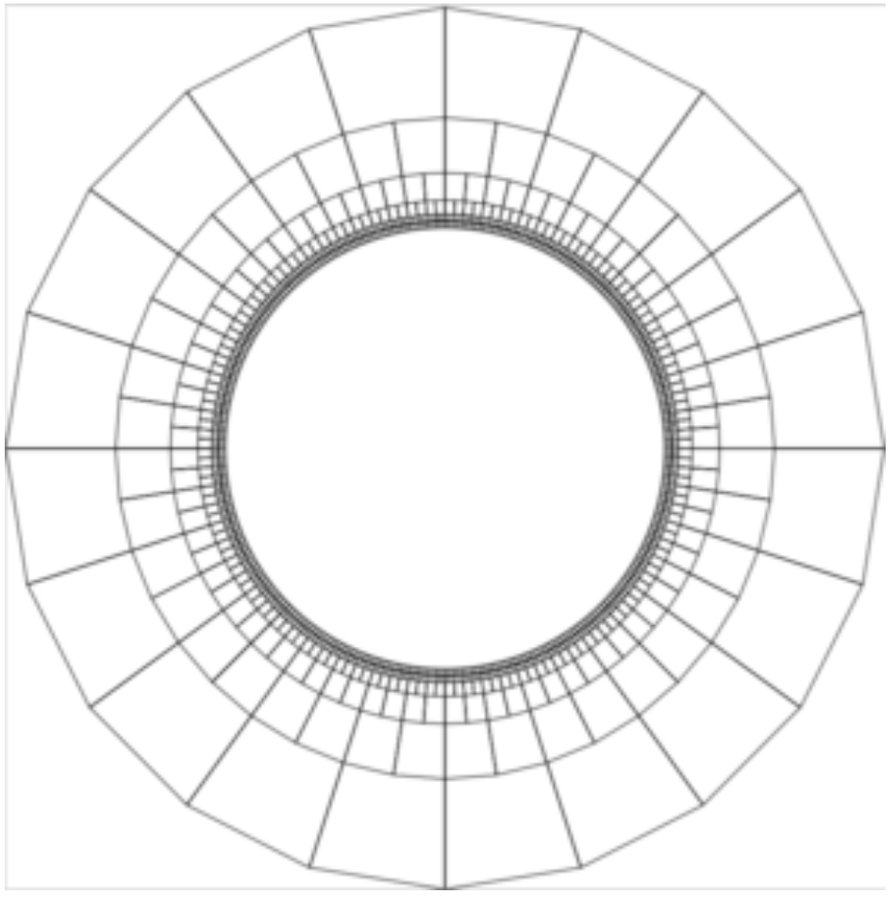
- Can enforce topologies
  - Manifolds on boundary
  - Internal manifolds
- Disadvantage
  - Cannot mix triangulation types
  - e.g. Volumetric body with extended manifold surface



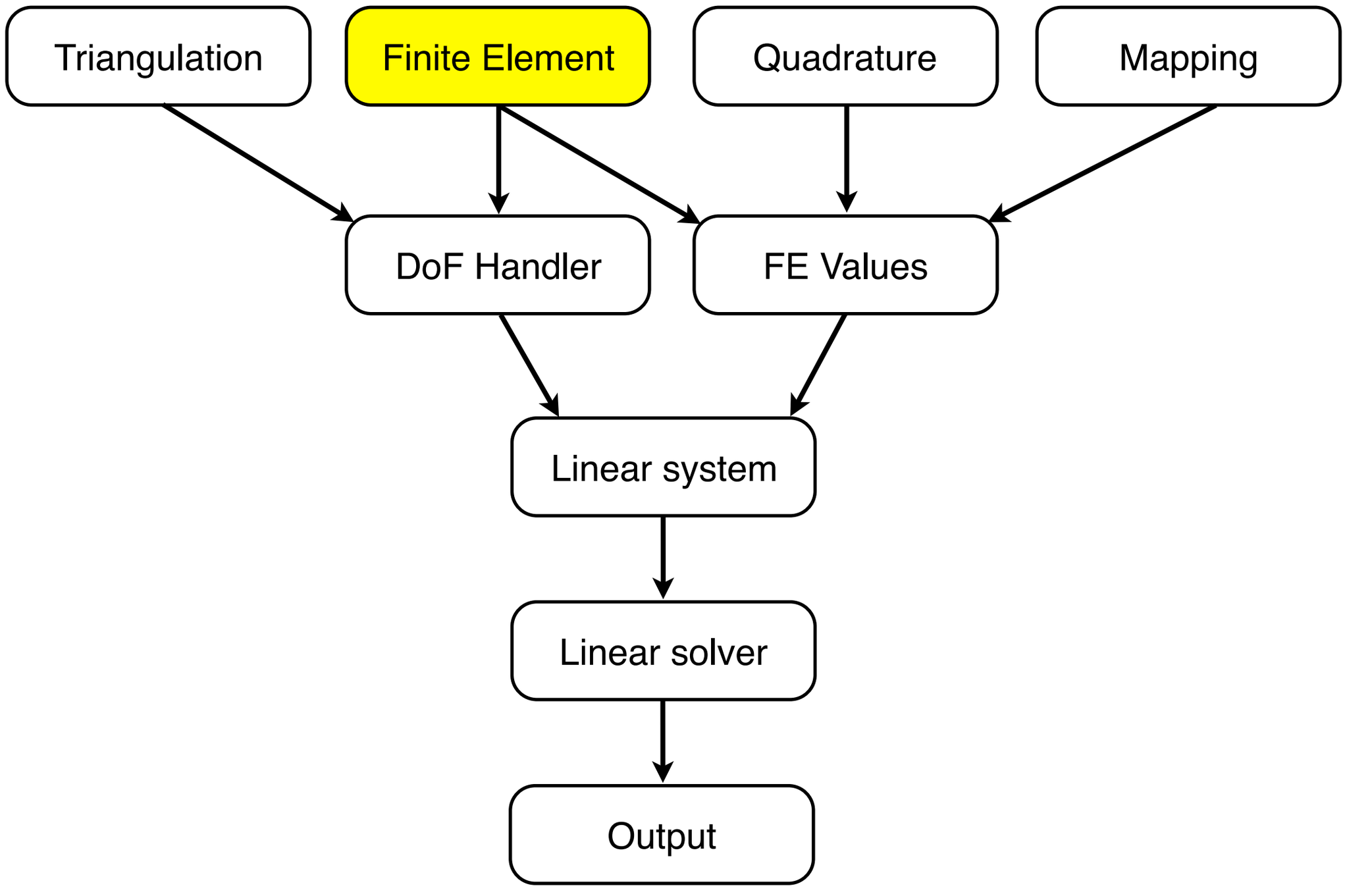
Interaction with geometry: the Triangulation class

- Demonstration: Step-1, step-49
  https://www.dealii.org/8.5.1/doxygen/deal.II/step_1.html
  https://www.dealii.org/8.5.1/doxygen/deal.II/step_49.html
  http://www.math.colostate.edu/~bangerth/videos.676.5.html
  http://www.math.colostate.edu/~bangerth/videos.676.6.html

- Key points
  - deal.II headers
  - Creating a triangulation
  - Boundary topology
  - Traversing a triangulation
  - Querying geometric information
  - Manipulating a triangulation
  - Aspects of grid refinement
  - Visualising a triangulation



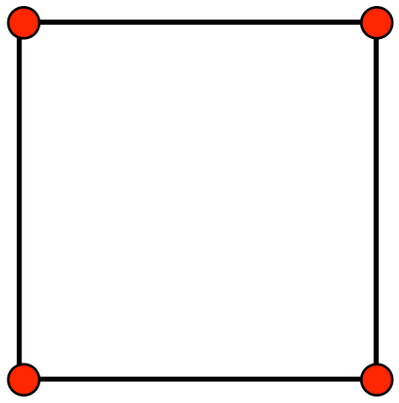
Structure of a prototypical FE problem



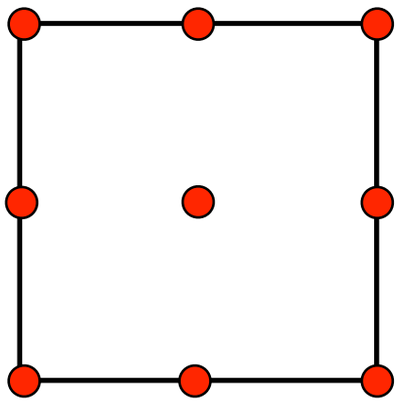
Assigning degrees-of-freedom: the FiniteElement classes

- Built in Finite Elements
  - Continuous
    - Piecewise Lagrange polynomials
  - Discontinuous
    - Monomials
    - Legendre polynomials
  - Vector-valued
    - Nedelec (H^{curl}, C/Dc)
    - Raviart-Thomas (H^{div}, C/Dc)
- A few more...
- Can develop finite elements from scratch
  - Specialisation for FE's derived by polynomial expansions
  - Enhanced/bubble elements

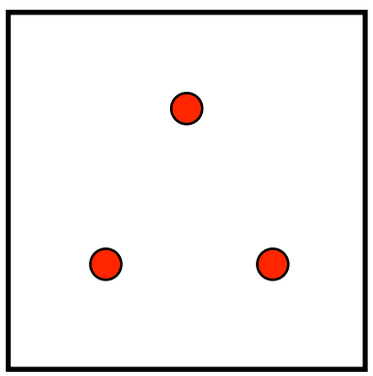
`FE_Q<2>(1)`



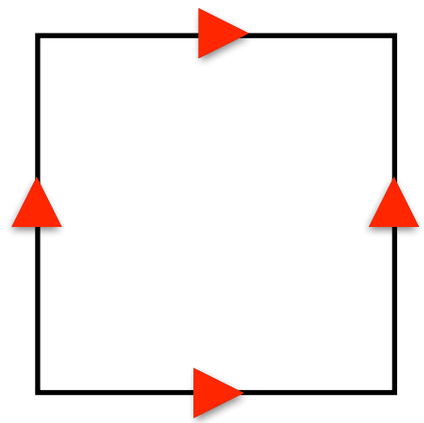
`FE_Q<2>(2)`



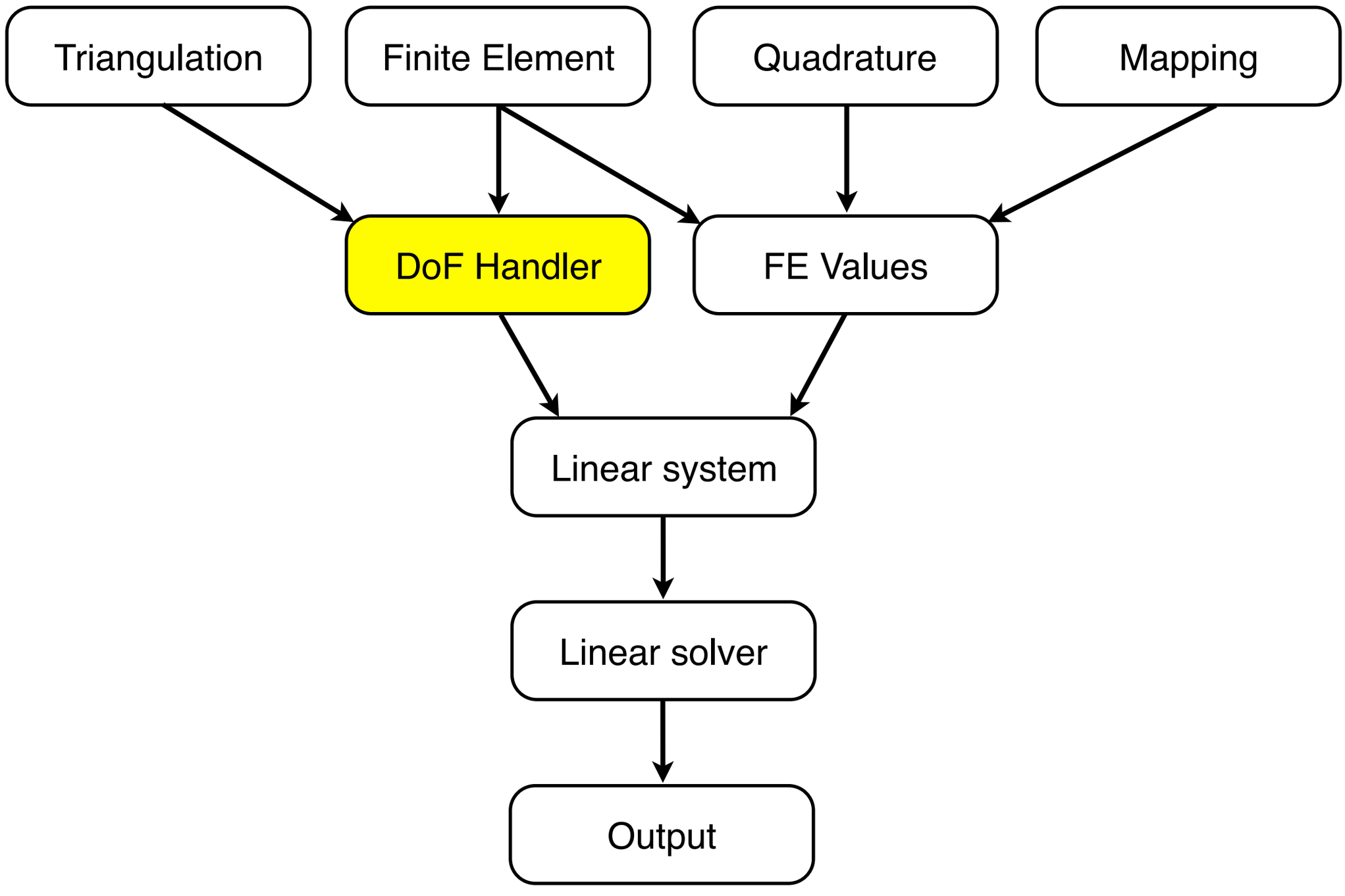
`FE_DGPMonomial<2>(1)`



`FE_Nedelec<2>(0)`

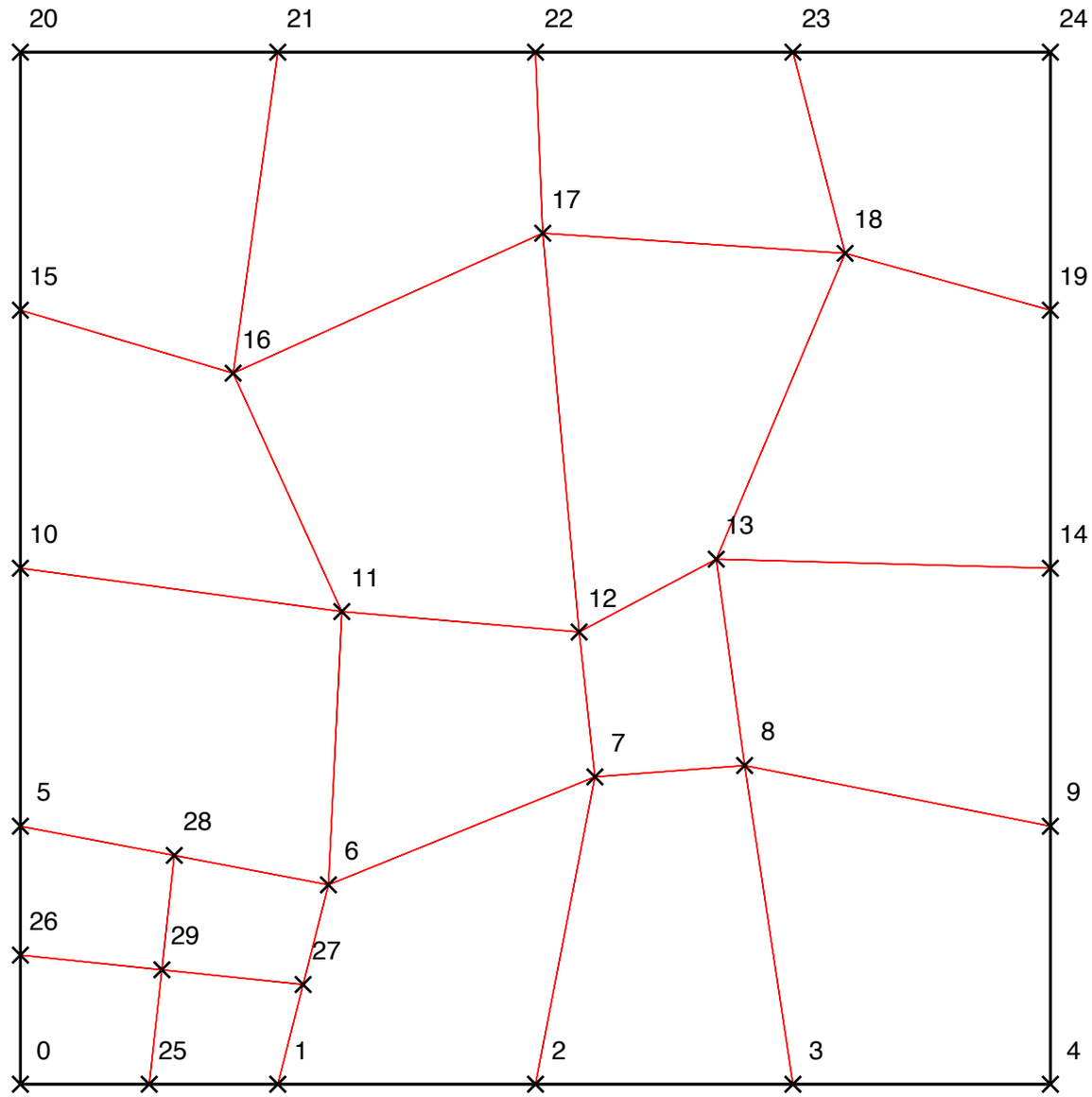


Structure of a prototypical FE problem



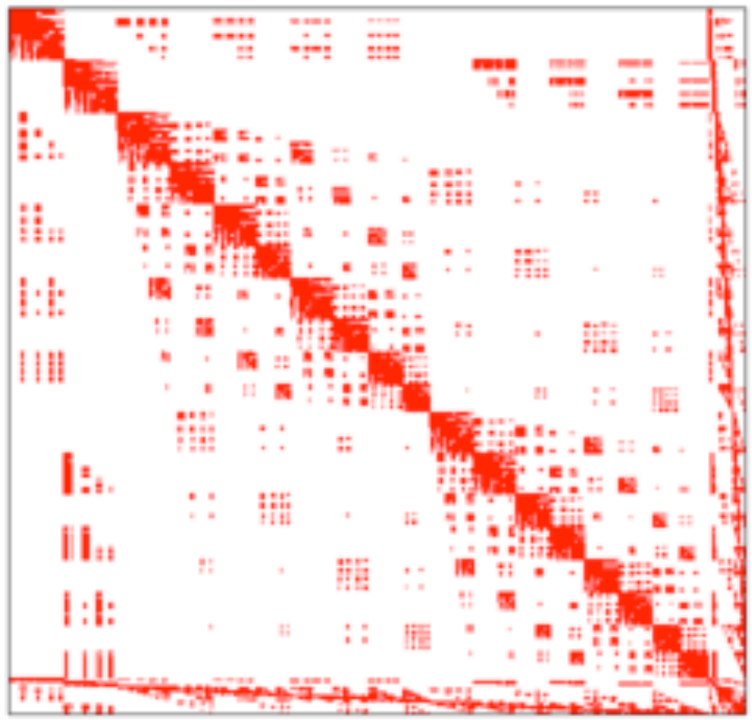
Assigning degrees-of-freedom: the DoFHandler class

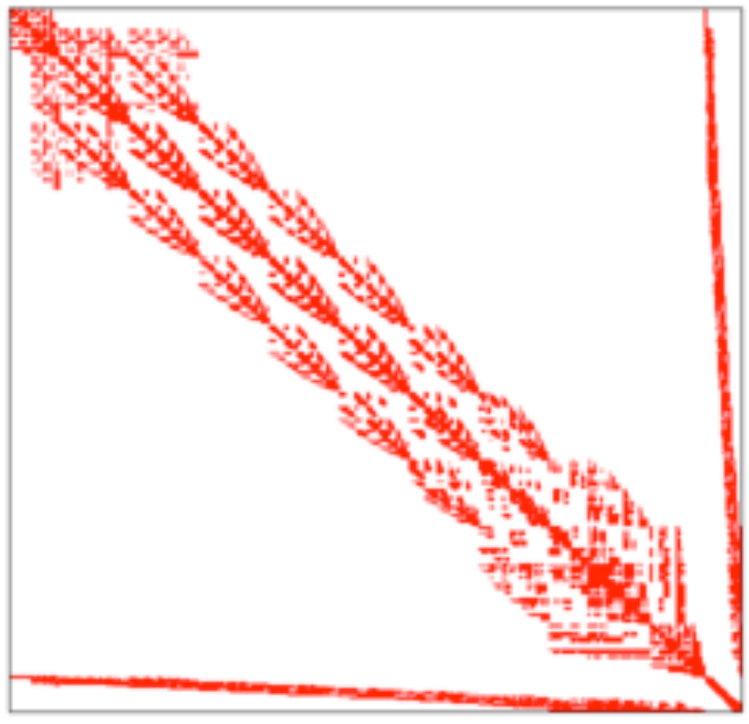
- DoFHandler assigns DoF's to grid
  - Important: separate to Triangulation!
- Unified way to access DoF's, regardless of FE used
  - e.g. Discontinuous elements: support points not necessarily at vertices
- Fast access and grid traversal
  - STL-type cell iterators
  - Access to faces, edges through these
- Disadvantage
  - All cells must have same types of finite-elements*



Assigning degrees-of-freedom: the DoFRenumbering namespace

- Renumbering schemes
  - Cuthill McKee
  - King
  - Downwind
- Reduce bandwidth
- Collect like-components
- Induce block-structure
- Directional (fluid flow)
- MPI subdomain



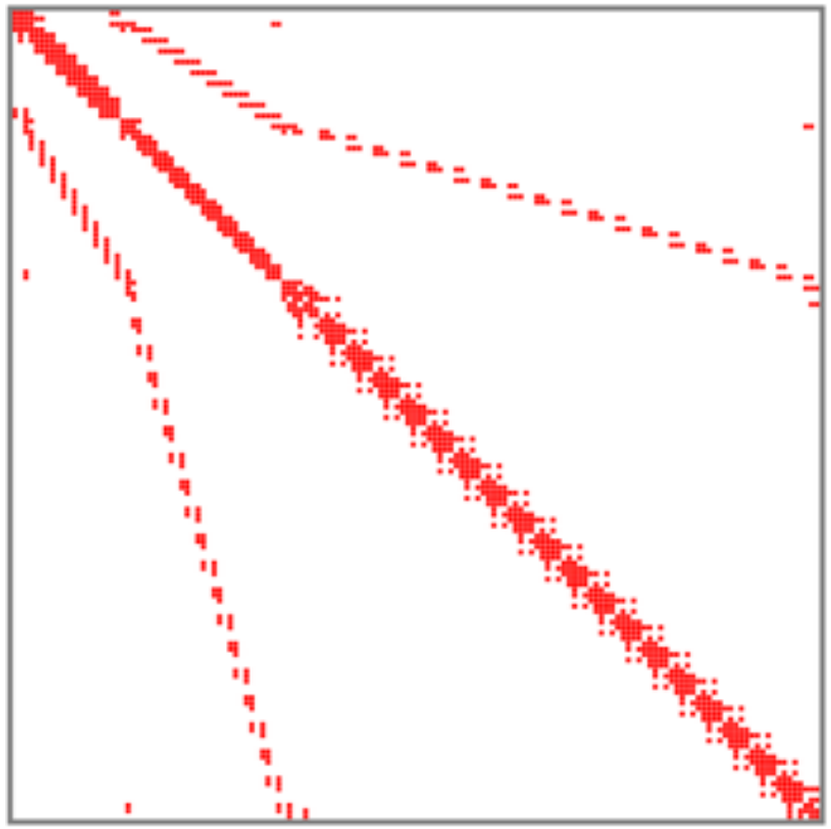


Assigning degrees-of-freedom: the FiniteElement and DoFHandler classes

- Demonstration: Step-2
  https://www.dealii.org/8.5.1/doxygen/deal.II/step_2.html
  http://www.math.colostate.edu/~bangerth/videos.676.9.html

- Key points

  - Choosing a Finite Element
  - Distributing degrees-of-freedom on a mesh
  - Renumbering degrees-of-freedom
  - Visualising sparsity patterns



A refresher on the Finite Element Method

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

19 March 2018

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Implementing the finite element method

Brief re-hash of the FEM, using the Poisson equation:

We start with the strong form:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned}$$

Implementing the finite element method

Brief re-hash of the FEM, using the Poisson equation:

We start with the strong form:

$$-\Delta u = f$$

...and transform this into the weak form by multiplying *from the left* with a test function:

$$(\nabla \varphi, \nabla u) = (\varphi, f) \quad \forall \varphi$$

The solution of this is a function *u(x)* from an infinite-dimensional function space.

Implementing the finite element method

Since computers can't handle objects with infinitely many coefficients, we seek a finite dimensional function of the form

$$u_h = \sum_{j=1}^{N} U_j \varphi_j(x)$$

To determine the *N* coefficients, test with the *N* basis functions:

$$(\nabla \varphi_i, \nabla u_h) = (\varphi_i, f) \quad \forall i = 1 \ldots N$$

If basis functions are linearly independent, this yields *N* equations for *N* coefficients.
This is called the *Galerkin* method.

Implementing the finite element method

Practical question 1: How to define the basis functions?

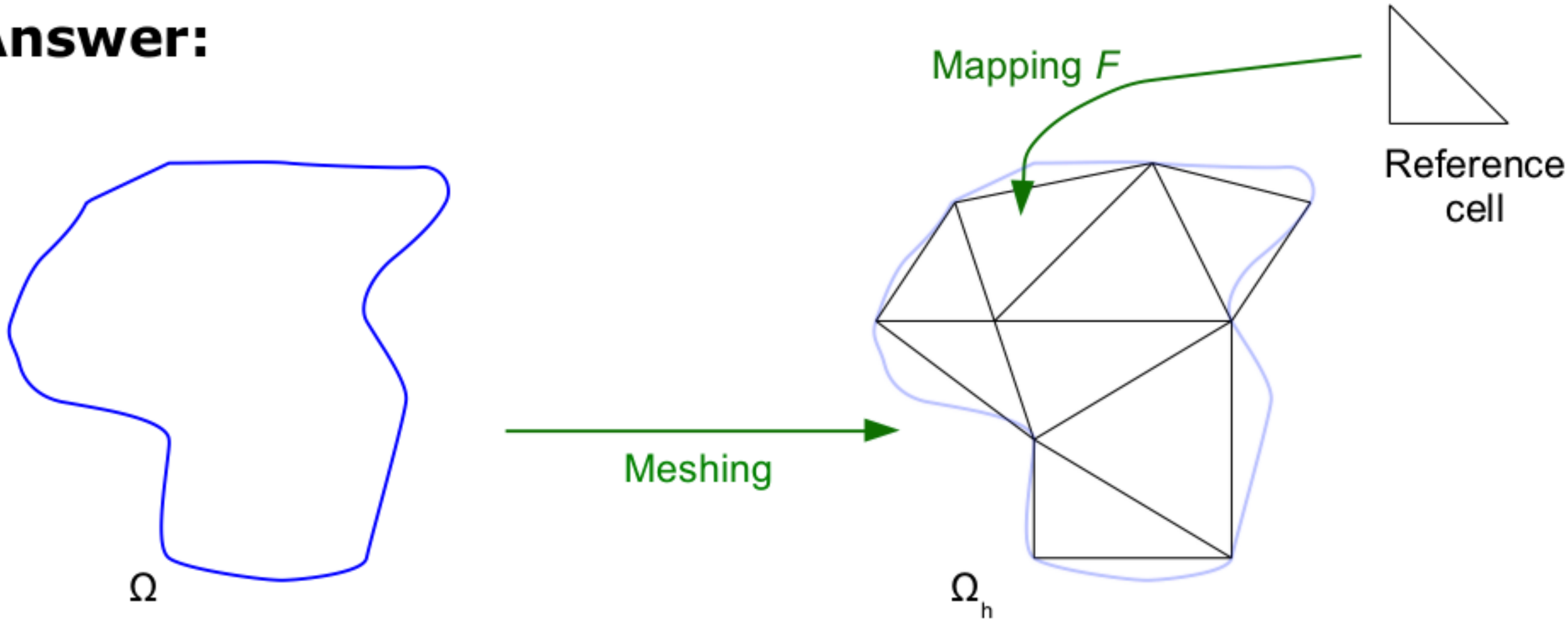
Answer: In the finite element method, this is done using the following concepts:

- Subdivision of the domain into a mesh
- Each cell of the mesh is a mapping of the reference cell
- Definition of basis functions on the reference cell
- Each shape function corresponds to a degree of freedom on the global mesh

Implementing the finite element method

Practical question 1: How to define the basis functions?

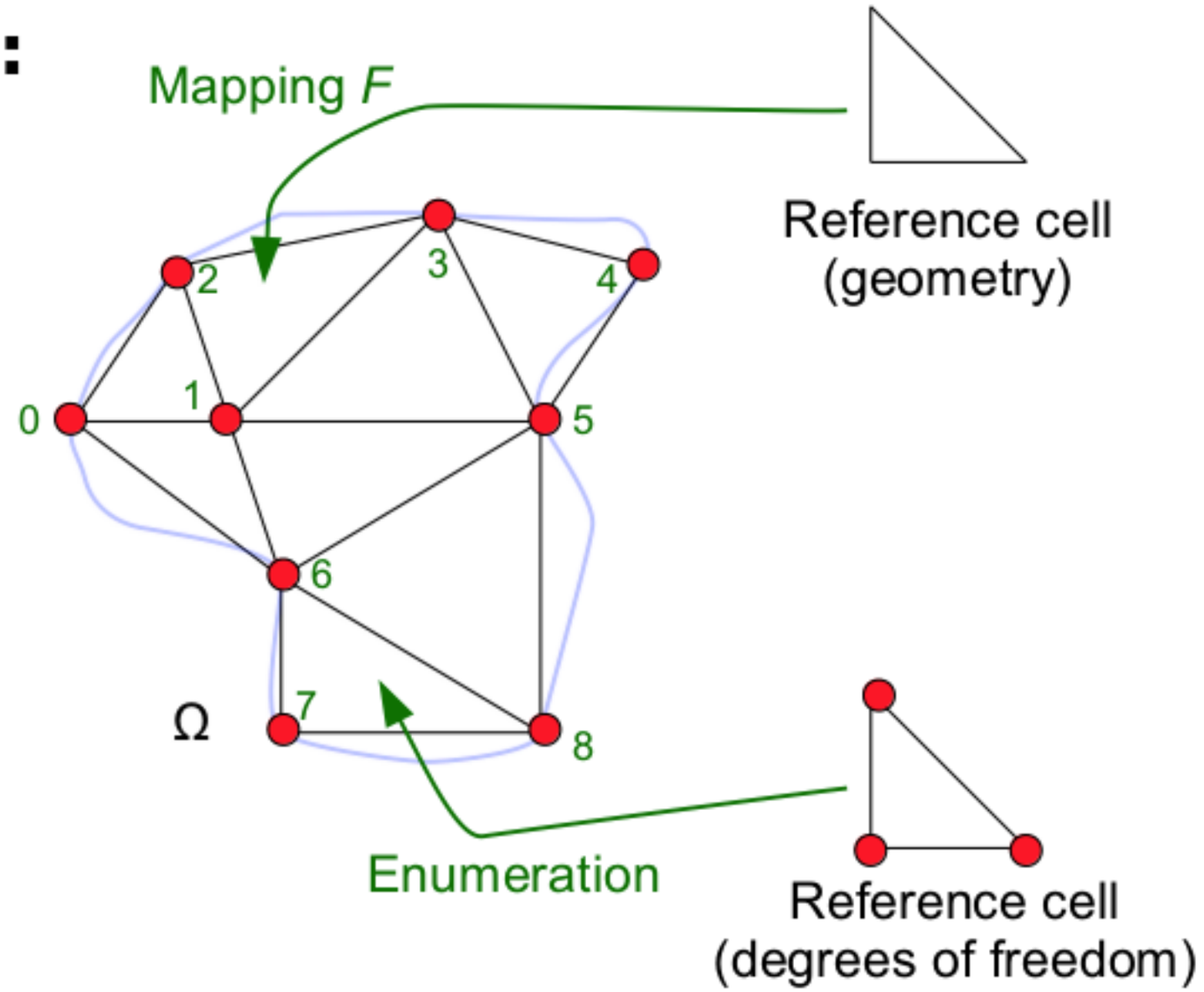
Answer:



Implementing the finite element method

Practical question 1: How to define the basis functions?

Answer:



Implementing the finite element method

Practical question 1: How to define the basis functions?

Answer: In the finite element method, this is done using the following concepts:

- Subdivision of the domain into a mesh
- Each cell of the mesh is a mapping of the reference cell
- Definition of basis functions on the reference cell
- Each shape function corresponds to a degree of freedom on the global mesh

Concepts in red will correspond to things we need to implement in software, explicitly or implicitly.

Implementing the finite element method

Given the definition $u_h = \sum_{j=1}^{N} U_j \varphi_j(x)$, we can expand the bilinear form

$$(\nabla \varphi_i, \nabla u_h) = (\varphi_i, f) \quad \forall i = 1 \ldots N$$

to obtain:

$$\sum_{j=1}^{N} (\nabla \varphi_i, \nabla \varphi_j) U_j = (\varphi_i, f) \quad \forall i = 1 \ldots N$$

This is a linear system

$$A U = F$$

with

$$A_{ij} = (\nabla \varphi_i, \nabla \varphi_j) \qquad F_i = (\varphi_i, f)$$

Implementing the finite element method

Practical question 2: How to compute

$$A_{ij}=(\nabla\varphi_i,\nabla\varphi_j) \qquad F_i=(\varphi_i,f)$$

Answer: By mapping back to the reference cell...

$$\begin{aligned} A_{ij} &= (\nabla\varphi_i,\nabla\varphi_j) \\ &= \sum_K \int_K \nabla\varphi_i(x)\cdot\nabla\varphi_j(x) \\ &= \sum_K \int_{\hat{K}} J_K^{-1}(\hat{x})\hat{\nabla}\hat{\varphi}_i(\hat{x}) \cdot J_K^{-1}(\hat{x})\hat{\nabla}\hat{\varphi}_j(\hat{x}) \ |\det J_K(\hat{x})| \end{aligned}$$

...and quadrature:

$$A_{ij} \approx \sum_K \sum_{q=1}^{Q} J_K^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_i(\hat{x}_q) \cdot J_K^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_j(\hat{x}_q) \ \underbrace{|\det J_K(\hat{x}_q)| \ w_q}_{=:\mathrm{JxW}}$$

Similarly for the right hand side *F*.

Implementing the finite element method

Practical question 3: How to store the matrix and vectors of the linear system

$$AU = F$$

Answers:

- *A* is sparse, so store it in compressed row format
- *U,F* are just vectors, store them as arrays
- Implement efficient algorithms on them, e.g. matrix-vector products, preconditioners, etc.
- For large-scale computations, data structures and algorithms must be parallel

Implementing the finite element method

Practical question 4: How to solve the linear system

$$AU = F$$

Answers: In practical computations, we need a variety of

- Direct solvers
- Iterative solvers
- Parallel solvers

Implementing the finite element method

Practical question 5: What to do with the solution of the linear system

$$AU = F$$

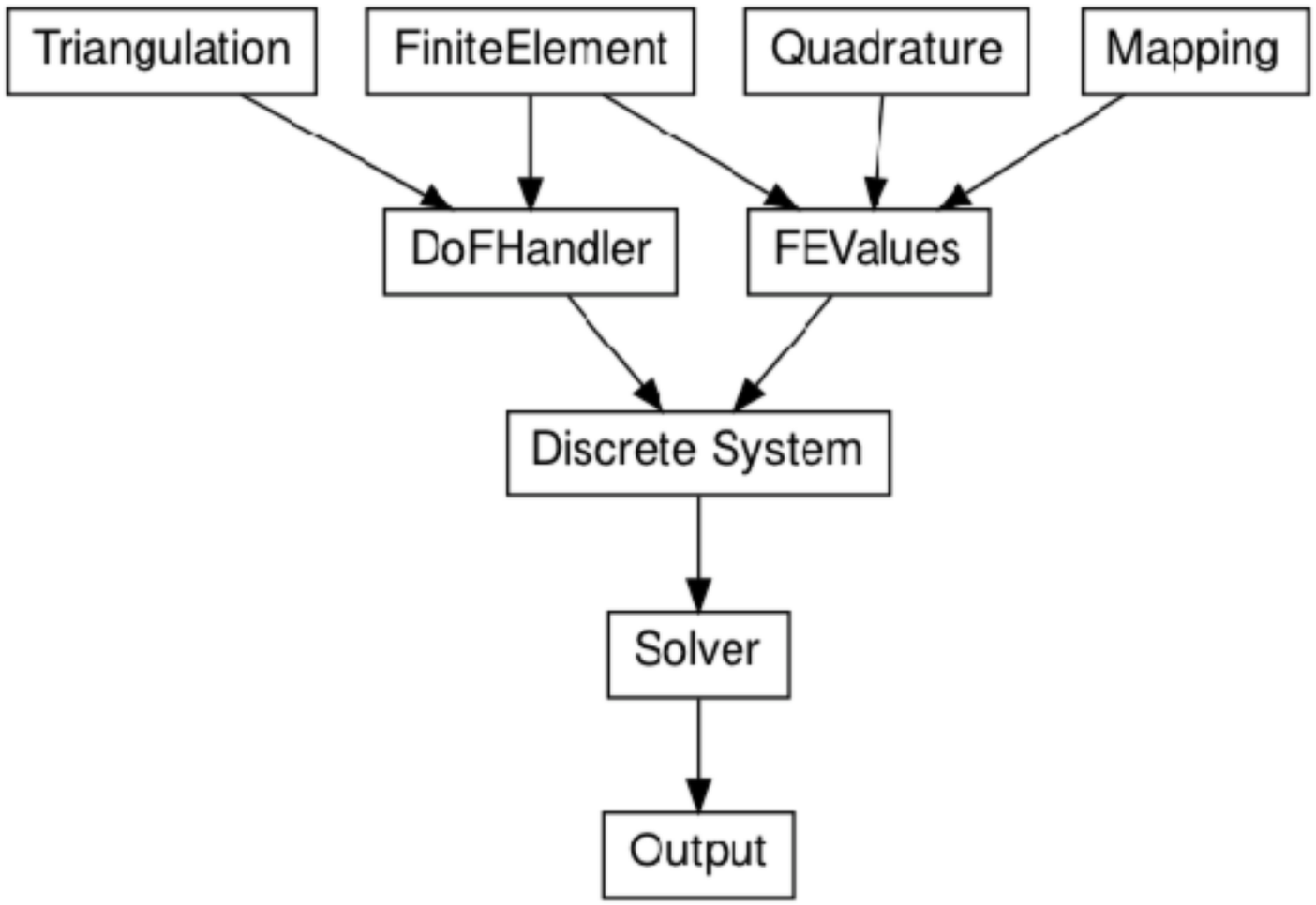
Answers: The goal is not to solve the linear system, but to do something with its solution:

- Visualize
- Evaluate for quantities of interest
- Estimate the error

These steps are often called *postprocessing the solution*.

Implementing the finite element method

Together, the concepts we have identified lead to the following components that all appear (explicitly or implicitly) in finite element codes:



Implementing the finite element method

Summary:

- By going through the mathematical description of the FEM, we have identified *concepts* that need to be represented by *software components*.

- Other components relate to what we *want to do* with numerical solutions of PDEs.

- The next few lectures will show the software realization of these concepts.

On using state-of-the-art tools

All of us have our favorite editor, often the first one we learned well:

- vi/vim/gvim
- emacs
- ...

But: Just because we know them well, this doesn't mean:

- That they are well suited to the task
- That they are state of the art
- That they are the tools that let you be most productive.

The rarest resource is *your* time, not CPU time etc.
You *must* be willing to keep learning new tools!

IDEs

All of us have our favorite editor, often the first one we learned well:

- vi/vim/gvim
- emacs
- ...

The problem with most of these:

- They are (good) editors but not code exploration tools
- They are text-based, not graphical

Excellent, modern tool are for example:

- eclipse
- kdevelop
- Xcode, Microsoft Visual C/C++

IDEs

What an IDE can do for you:

IDEs "know" about your code base, i.e., they *parse all* of the files that belong to your project.

Thus, the IDE...

- Knows where a variable is declared (even if in a different file)
- Knows its type and can help you with *code completion*
- Can *rename* a variable everywhere
- Can keep declaration and definition in sync
- Can help you with function arguments
- Makes you *faster* and make *far fewer mistakes.*

Solving Poisson's equation

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

19 March 2018

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

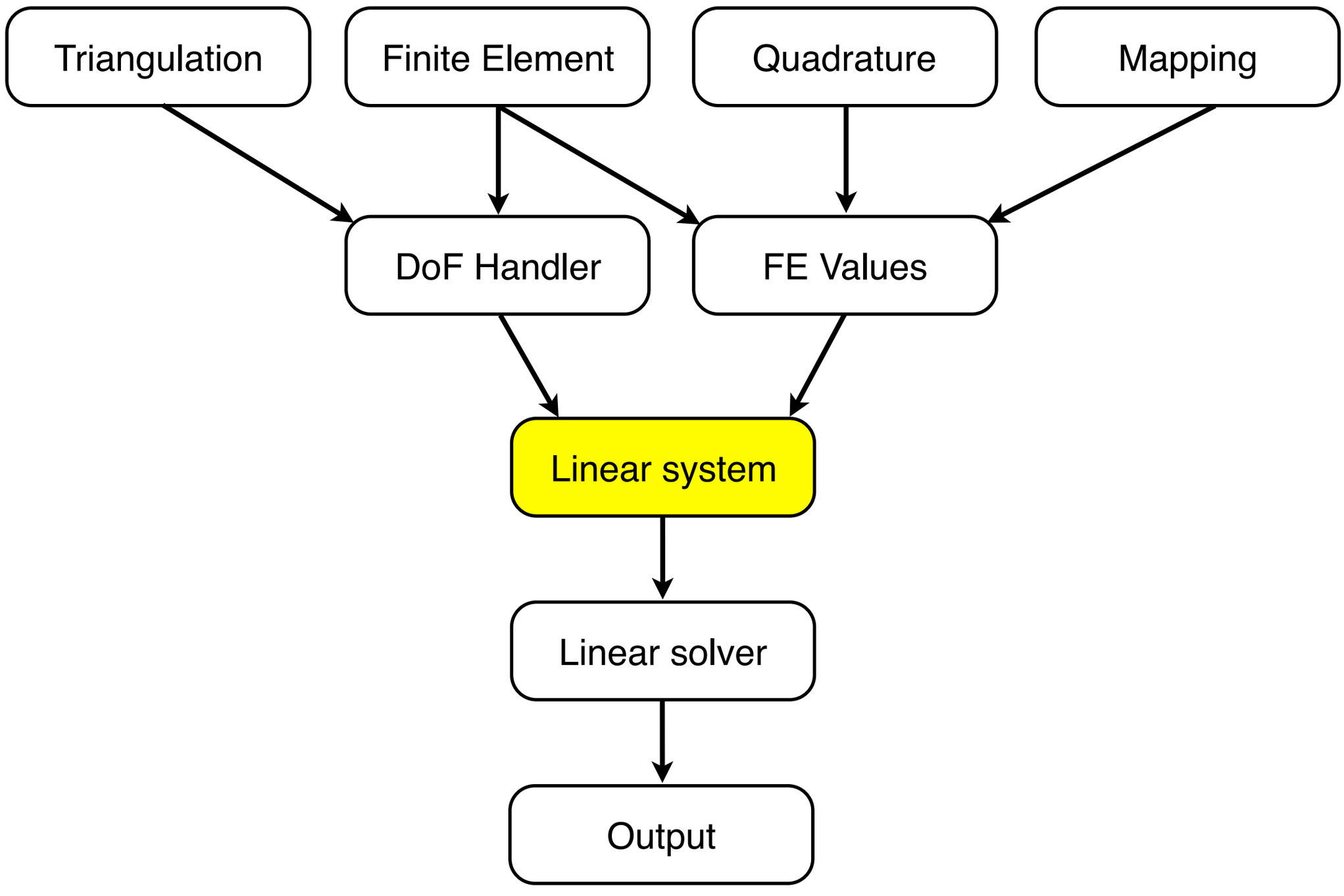
Aims for this module

- First introduction into assembly of sparse linear systems
  - Translation of weak form to assembly loops
  - Applying boundary conditions
- Using linear solvers
- Post-processing and visualisation

Reference material

- Tutorials
  - Step-3 https://dealii.org/8.5.1/doxygen/deal.II/step_3.html
- Documentation
  - https://www.dealii.org/developer/doxygen/deal.II/group__FE__vs__Mapping__vs__FEValues.html
  - https://www.dealii.org/developer/doxygen/deal.II/group__UpdateFlags.html

Structure of a prototypical FE problem



Sparse linear systems

- Minimise data storage
  - Evaluate grid connectivity
- Functions to help set up
  - Connectivity
  - Constraints
- Minimal access times
  - Direct manipulation of (non-zero) entries
  - Matrix-vector operations
    - Skip over zero-entries
- Types
  - Unity (monolithic, contiguous)
  - Block sparse structures
- Sub-organisation (e.g. component-wise)

$$[K]\{d\} = \{F\}$$

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \end{Bmatrix}$$

- $$\left(K_{11} - K_{12}K_{22}^{-1}K_{21}\right) d_1 = F_1 - K_{12}K_{22}^{-1}F_2$$
- $$d_2 = K_{22}^{-1}\left(F_2 - K_{21}d_1\right)$$

Constraints on sparse linear systems

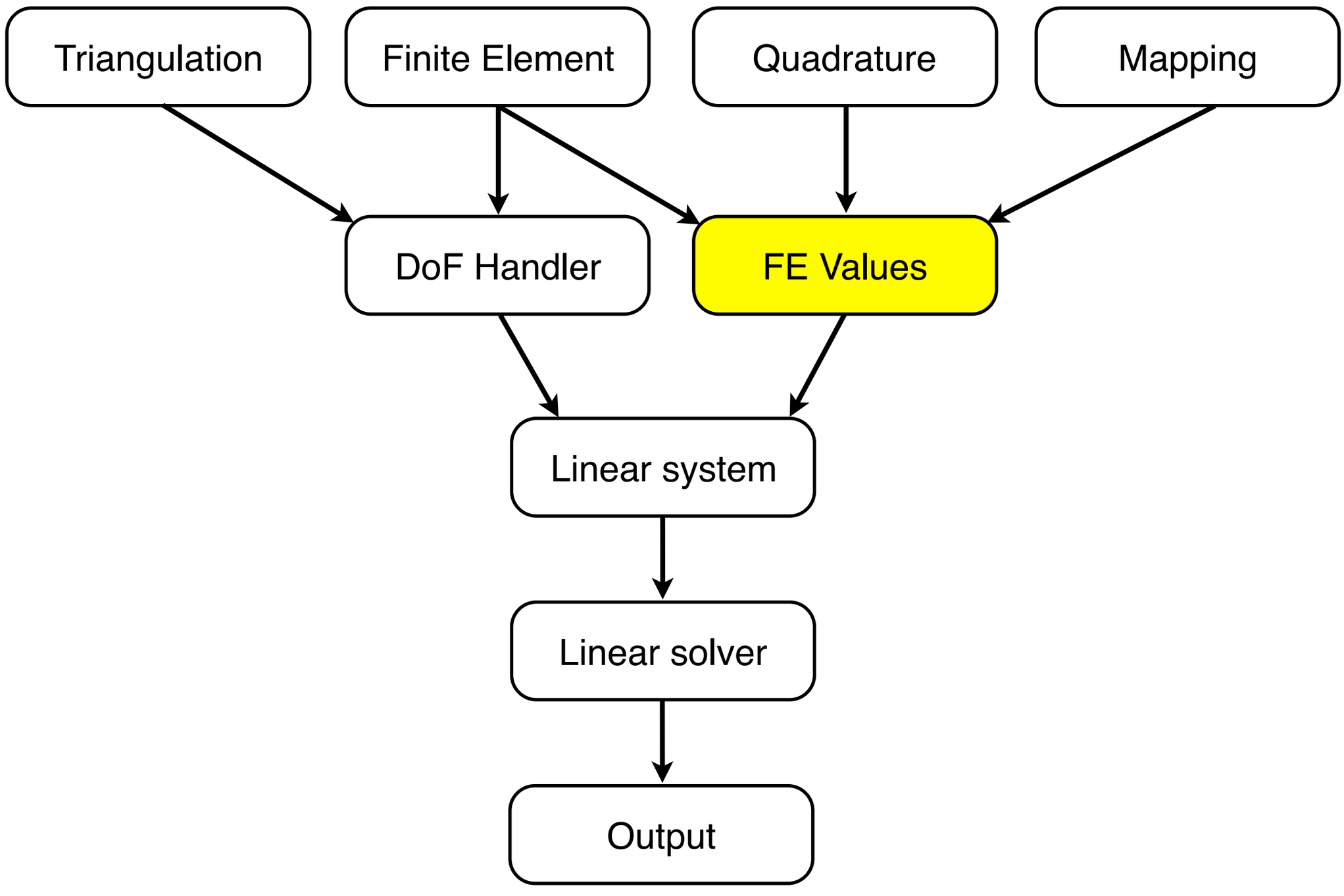
- Strong Dirichlet boundary conditions
  - Apply user-defined spatially-dependent functions to specific boundaries
  - Can restrict to components of a multi-dimensional field
  - Limited to fields with support points on faces
    - Possible to scale system matrix/RHS vector accordingly
      - Better matrix conditioning
- Neumann boundary conditions
  - Implementation dependent
- Other constraints need special consideration
  - Periodic boundary conditions
  - Refinement with hanging nodes
  - Some time-dependent formulations

$$[K]\{d\} = \{F\}$$

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{Bmatrix} d_1 \\ d_2 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ F_2 \end{Bmatrix}$$

- $\left(K_{11} - K_{12}K_{22}^{-1}K_{21}\right) d_1 = F_1 - K_{12}K_{22}^{-1}F_2$
- $d_2 = K_{22}^{-1}\left(F_2 - K_{21}d_1\right)$

Structure of a prototypical FE problem



Integration on a cell: the FEValues class

$$K = \int_\Omega \nabla \delta\phi \cdot k \nabla \phi dV$$

$$\approx \delta\phi^I \sum_K \left(\int_{\Omega^h} \nabla N^I \cdot k \nabla N^J dV^h \right) \phi^J$$

$$\approx \delta\phi^I \sum_K \underbrace{\left(\sum_q \nabla N^I(\mathbf{x}_q) \cdot k_q \nabla N^J(\mathbf{x}_q)\, w_q \right)}_{K_{IJ} = (\nabla\phi^I, k\nabla\phi^J)} \phi^J$$

$$\approx \delta\phi^I \underbrace{\sum_K \left(\sum_q \left[J_\square^K\right]_q^{-1} \nabla_\square N^I(\mathbf{x}_q) \cdot k_q \left[J_\square^K\right]_q^{-1} \nabla_\square N^J(\mathbf{x}_q)\, |\det J_q| w_q \right)}_{K_{IJ}} \phi^J$$

$$J_\square^h = \frac{\partial \mathbf{X}^{\boldsymbol{\xi}}}{\partial \mathbf{X}}$$

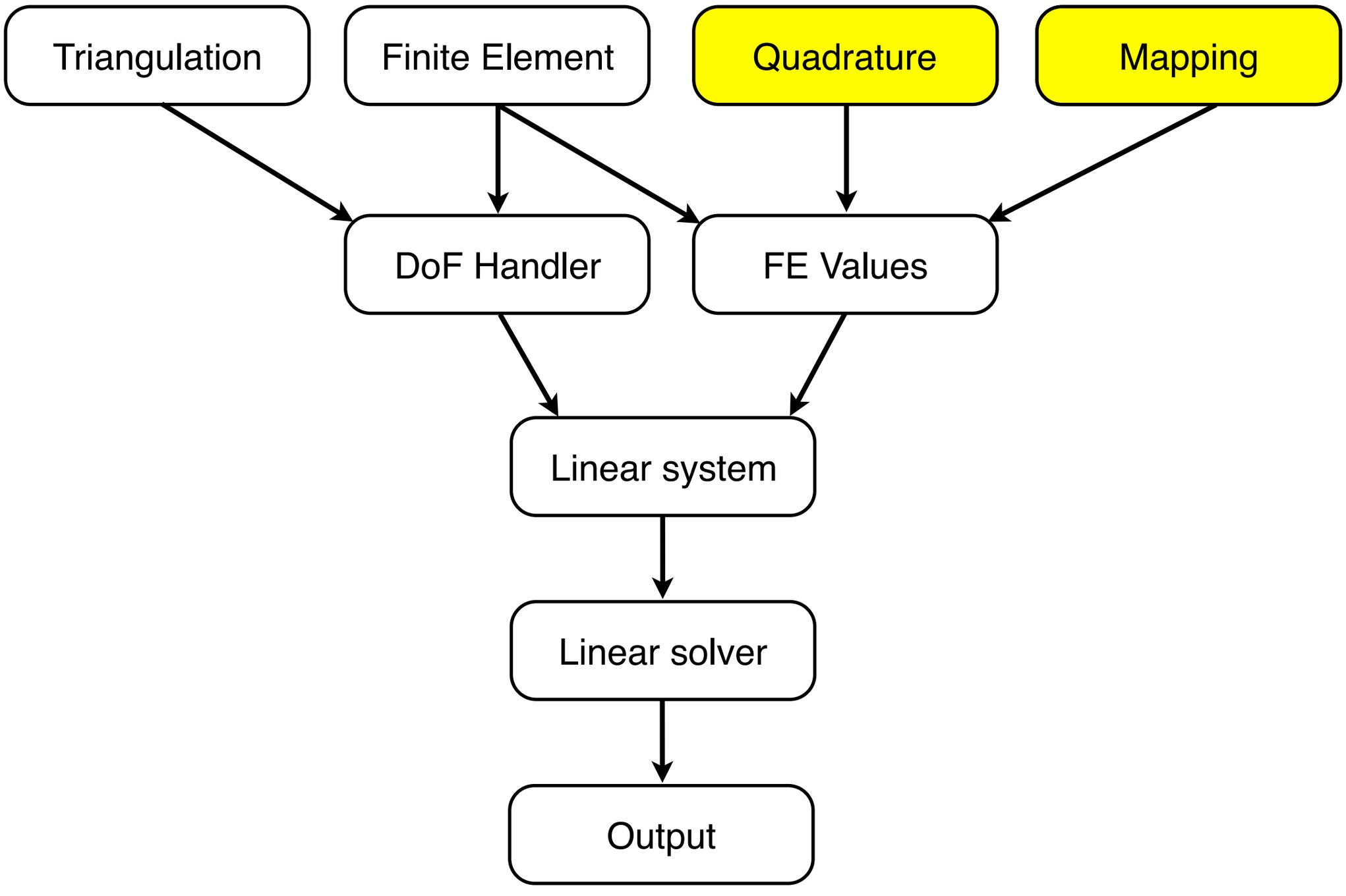
Integration on a cell: the FEValues class

- Object that helps perform integration
- Combines information of:
  - Cell geometry
  - Finite-element system
  - Quadrature rule
  - Mappings
- Can provide:
  - Shape function data
  - Quadrature weights and mapping jacobian at a point
  - Normal on face surface
  - Covariant/contravariant basis vectors
- More ways it can help:
  - Object to extract shape function data for individual fields
  - Natural expressions when coding
- Low level optimisations

$$K_{IJ} = \sum_K \left(\sum_q \left[J_\square^K \right]_q^{-1} \nabla_\square N^I \left(\mathbf{x}_q \right) \cdot k_q \left[J_\square^K \right]_q^{-1} \nabla_\square N^J \left(\mathbf{x}_q \right) \left| \det J_q \right| w_q \right)$$

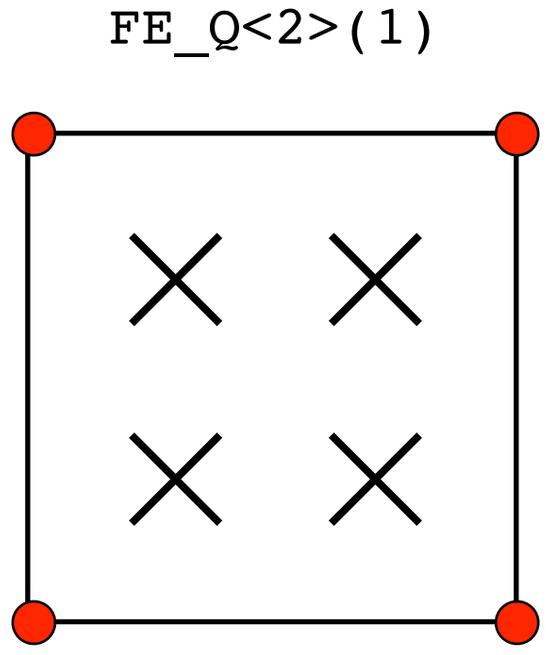
```
cell_matrix(I,J) += k
      * fe_values.shape_grad (I, q_point)
      * fe_values.shape_grad (J, q_point)
      * fe_values.JxW (q_point);
```

Structure of a prototypical FE problem



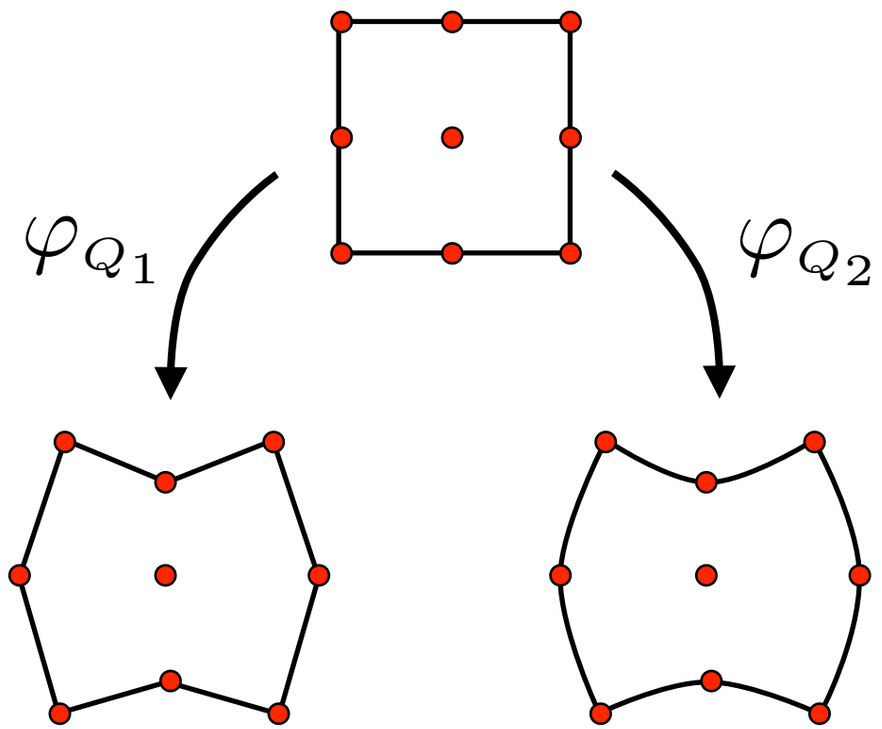
Integration on a cell: the Quadrature classes

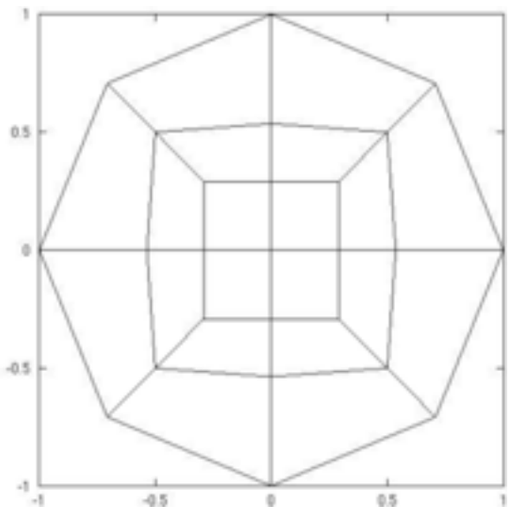
- n-Order Gauss quadrature
- Other rules
  - Gauss Lobatto
  - Simpson
  - Trapezoidal
  - Midpoint
  - A few others
- Anisotropic
  - Tensor product

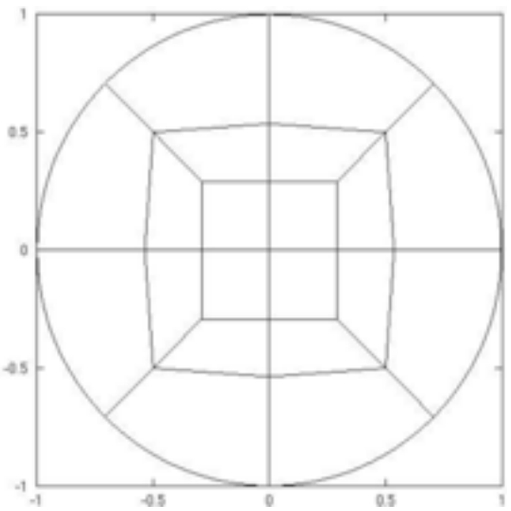


Integration on a cell: the Mapping classes

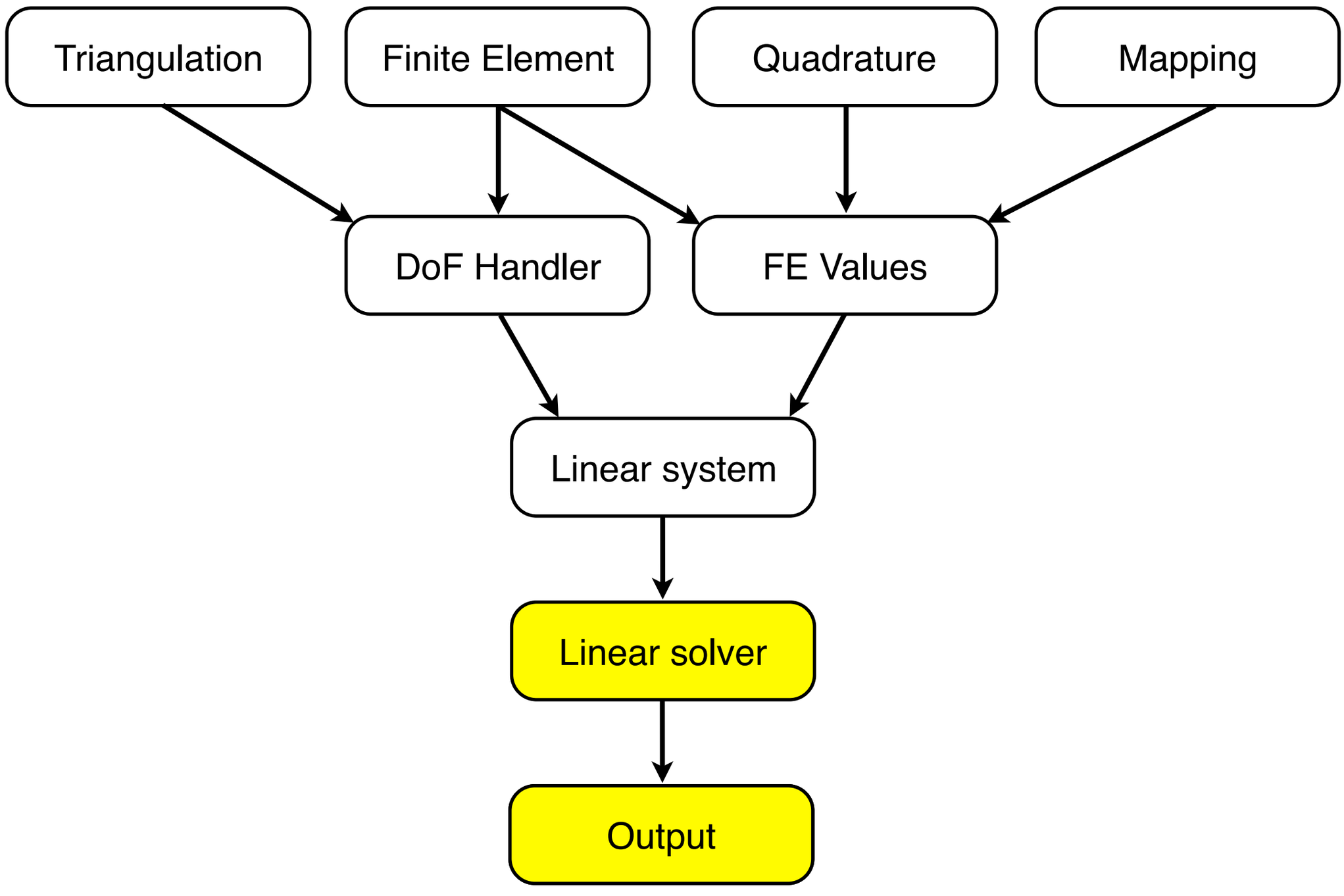
- n-order mappings
  - Increase accuracy of:
    - Integration schemes
    - Surface basis vectors
- Lagrangian / Eulerian
  - Latter useful for fluid and contact problems, data visualisation
- Boundary and interior manifolds







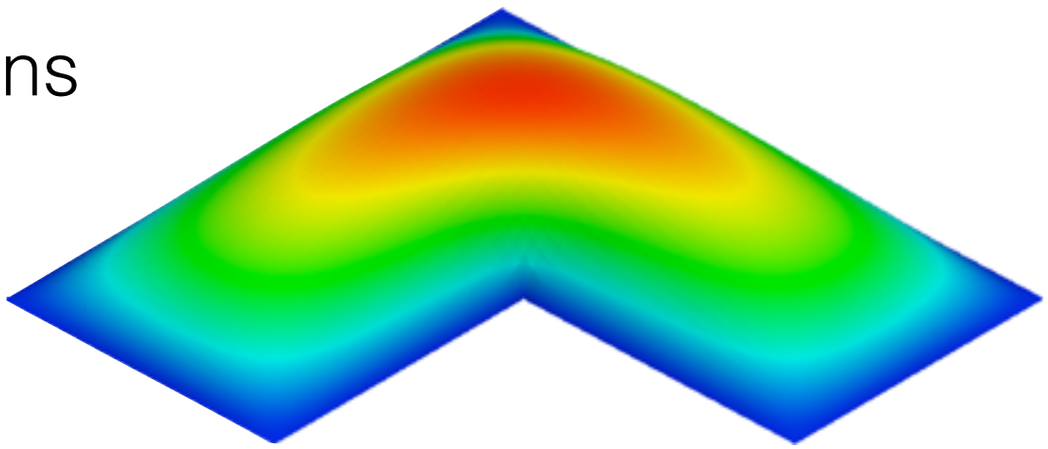
Structure of a prototypical FE problem



Solving Poisson's equation

- Demonstration: Step-3
  https://www.dealii.org/8.5.1/doxygen/deal.II/step_3.html
  http://www.math.colostate.edu/~bangerth/videos.676.10.html

- Key points
  - Local assembly + quadrature rules
  - Distribution of local contributions to the global linear system
  - Application of boundary conditions
  - Solving a linear system
  - Output for visualisation



Global and local error computation and estimation

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

19 March 2018

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Elements

There is a zoo of elements for different purposes:

- Continuous Lagrange
- Discontinuous Lagrange
- Raviart-Thomas
- Nedelec
- Rannacher-Turek
- Brezzi-Douglas-Marini (BDM)
- Brezzi-Douglas-Duran-Marini (BDDM)
- Hermite (Argyris)
- Crouzeix-Raviart
- Arnold-Falk-Winther
- Arnold-Boffi-Falk (ABF)

  ...
- Hybridized elements
- Penalized discontinuous elements

Elements

There is a zoo of elements for different purposes:

- Continuous Lagrange — FE_Q
- Discontinuous Lagrange — FE_DGQ, FE_DGP
- Raviart-Thomas — FE_RaviartThomas
- Nedelec — FE_Nedelec
- Rannacher-Turek
- Brezzi-Douglas-Marini (BDM) — FE_BDM
- Brezzi-Douglas-Duran-Marini (BDDM)
- Hermite (Argyris)
- Crouzeix-Raviart
- Arnold-Falk-Winther
- Arnold-Boffi-Falk (ABF) — FE_ABF

  ...
- Hybridized elements — FE_FaceQ/TraceQ
- Penalized discontinuous elements

Scalar problems

For scalar problems like the Laplace equation:

- Q_p elements are generally the right choice
- Higher *p* yield higher convergence order for elliptic problems:

$$\|u-u_h\|_{H^1} \leq Ch^p|u|_{H^{p+1}} \qquad \|u-u_h\|_{L_2} \leq Ch^{p+1}|u|_{H^{p+1}}$$

- Number of degrees of freedom grows as:

$$N \simeq \frac{|\Omega|}{(h/p)^d} = p^d\frac{|\Omega|}{h^d} \qquad \rightarrow \qquad h \simeq p\left(\frac{|\Omega|}{N}\right)^{1/d}$$

- Error as function of *N*:

$$\|u-u_h\|_{H^1} \simeq p^p N^{-p/d} \qquad \|u-u_h\|_{L_2} \simeq p^{p+1} N^{-(p+1)/d}$$

Consequence: This suggests high order elements!

Scalar problems

For scalar problems like the Laplace equation:

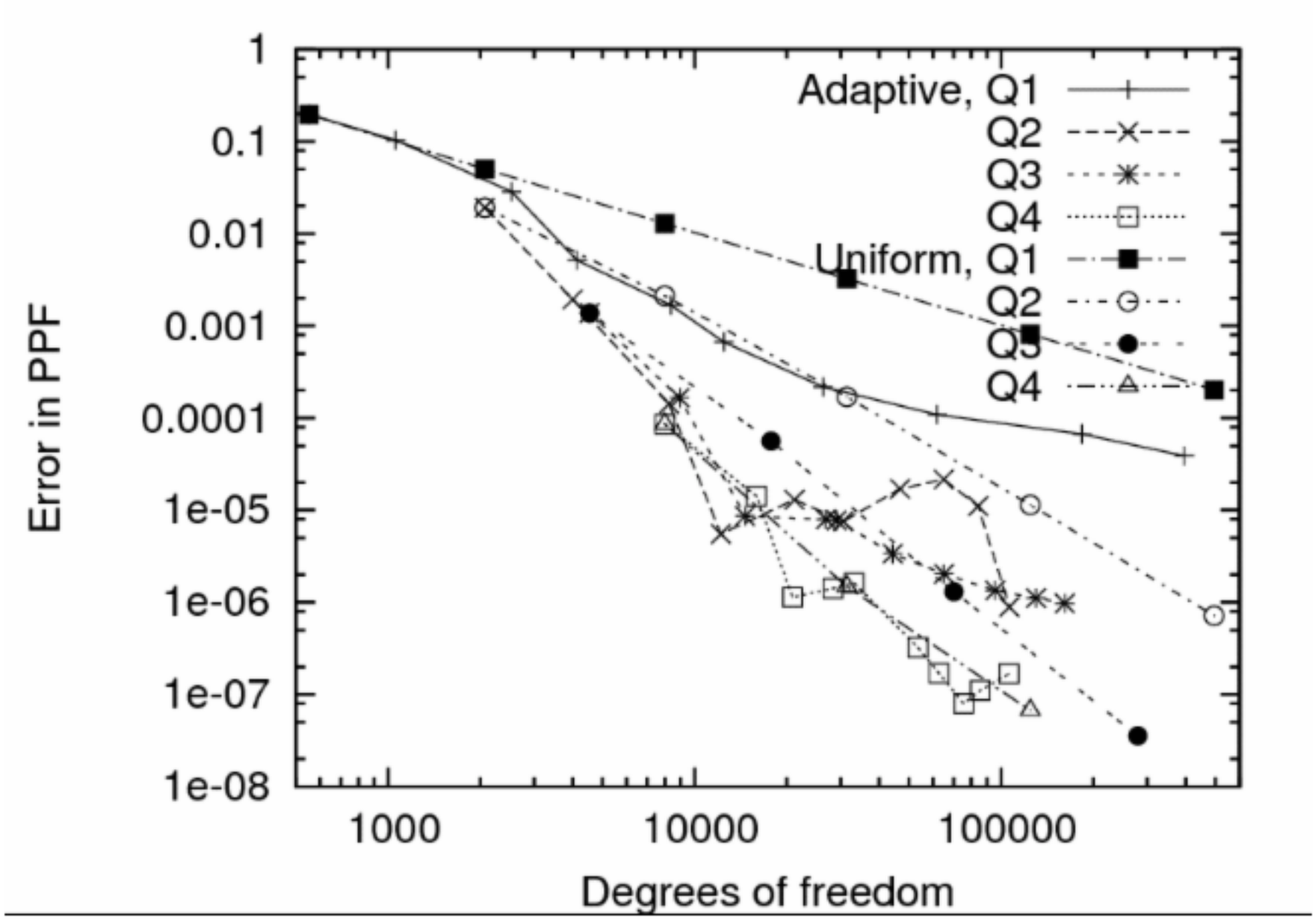
- Q_p elements are generally the right choice
- Better convergence only if *u* smooth: $\|u-u_h\|_{H^1} \leq Ch^p|u|_{H^{p+1}}$
- Higher *p* also requires more work:
  - more computations to assemble matrix: $O(p^d)$
  - more entries per row in the matrix: $O(p^d)$
  - good preconditioners are difficult to construct for high *p*

Consequence: This suggests low order elements!

Together: It is a trade-off!

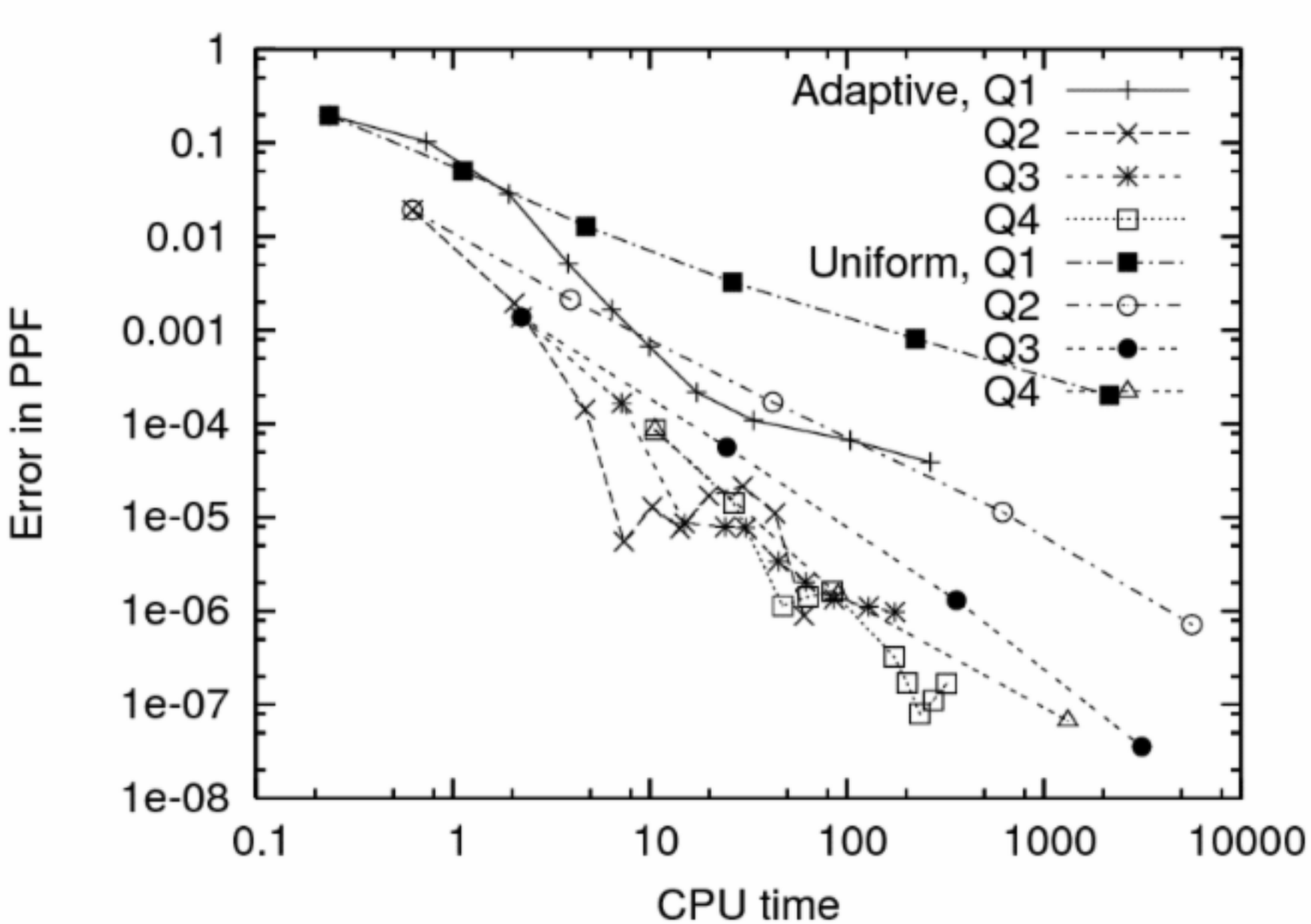
Practical experience

Prototypical 2d example from Wang, Bangerth, Ragusa (2007, Progress in Nuclear Energy):



Practical experience

Prototypical 2d example from Wang, Bangerth, Ragusa (2007, Progress in Nuclear Energy):



Practical experience

Prototypical 2d example from Wang, Bangerth, Ragusa (2007, Progress in Nuclear Energy):

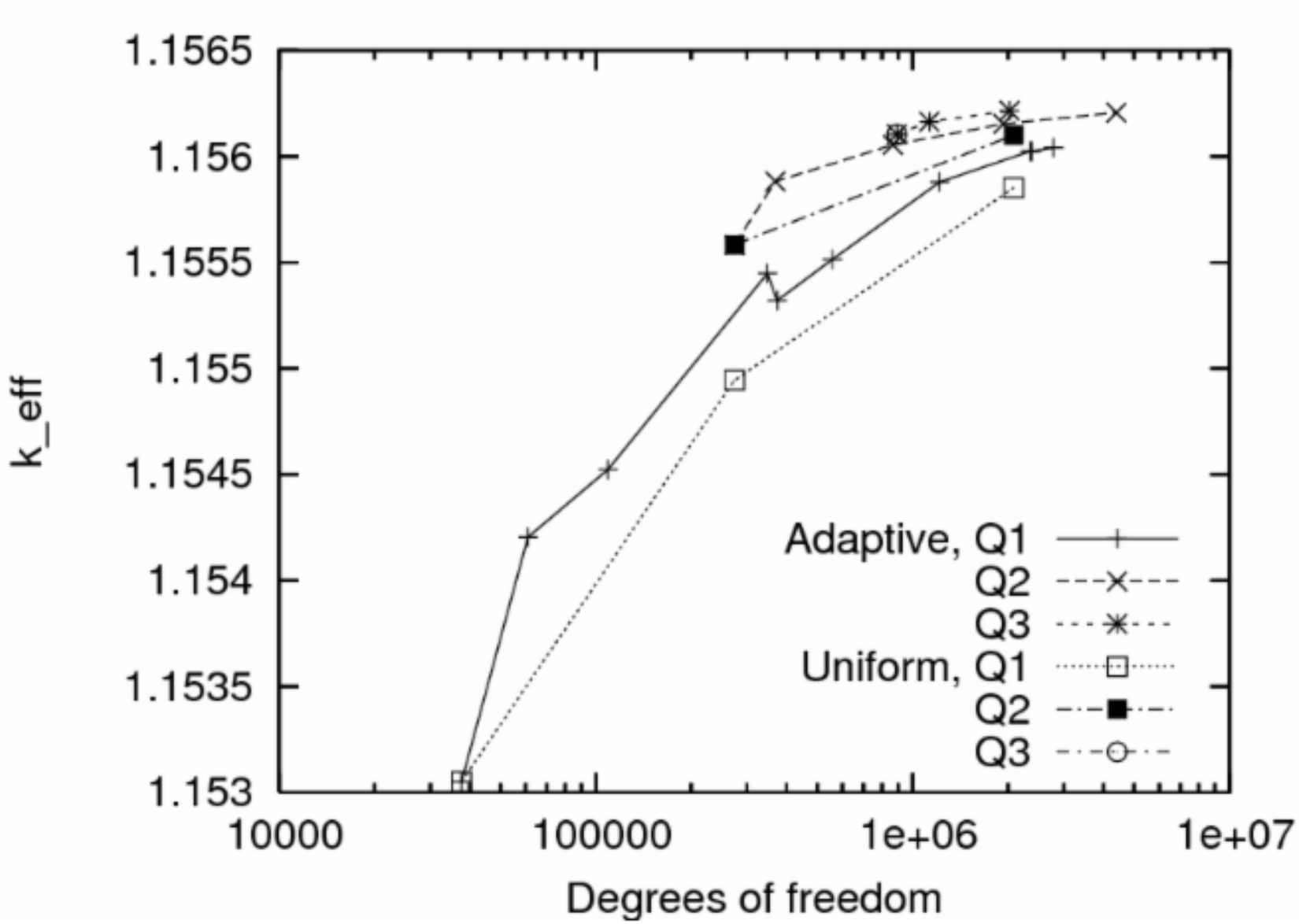
Conclusions:

- Higher *p* gives better error-per-dof
- Not so clear any more for error-per-CPU-second

- Sweat spot maybe around *p=3* or *p=4* in 2d

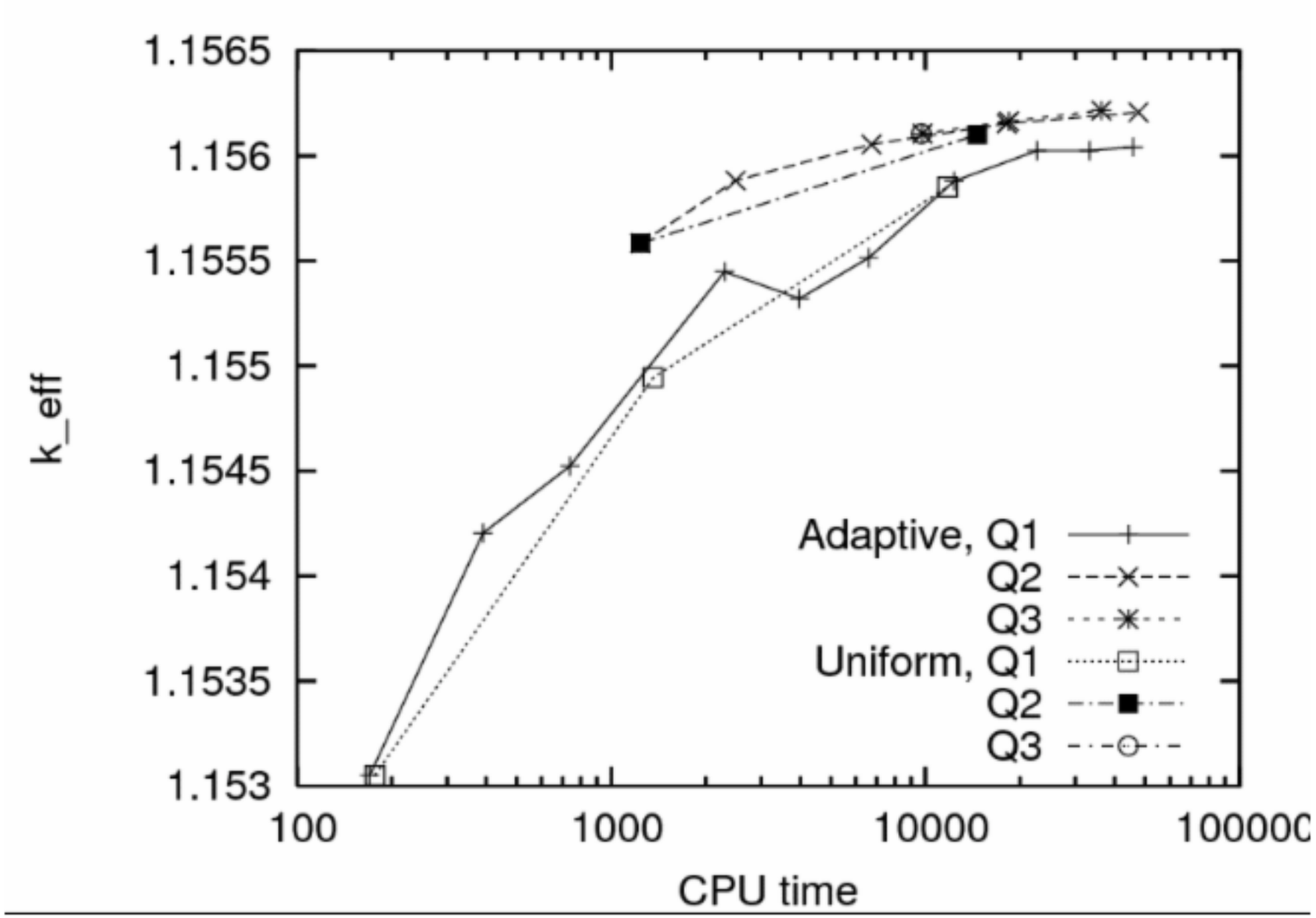
Practical experience

Prototypical 3d example from Wang, Bangerth, Ragusa (2007, Progress in Nuclear Energy):



Practical experience

Prototypical 3d example from Wang, Bangerth, Ragusa (2007, Progress in Nuclear Energy):



Practical experience

Prototypical 3d example from Wang, Bangerth, Ragusa (2007, Progress in Nuclear Energy):

Conclusions:

- Higher *p* gives better error-per-dof
- Not so clear any more for error-per-CPU-second

- Sweat spot maybe around *p=2* or *p=3* in 3d

Practical experience

Conclusions for scalar problems:

- There is a trade-off between faster convergence and more work

- A good compromise is:
  - Q3 or Q4 in 2d
  - Q2 or Q3 in 3d

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)
  - Possibly integrate the difference between the gradients of the exact and computed solutions

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

Estimate the rate of convergence

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

$$\|u - u_h\|_1 \leq Ch^k |u|_{k+1}$$

$$\|u - u_h\|_0 \leq Ch^{k+1} |u|_{k+1}$$

Estimate the rate of convergence

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

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

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

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

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

Adaptive mesh refinement (AMR)

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

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

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

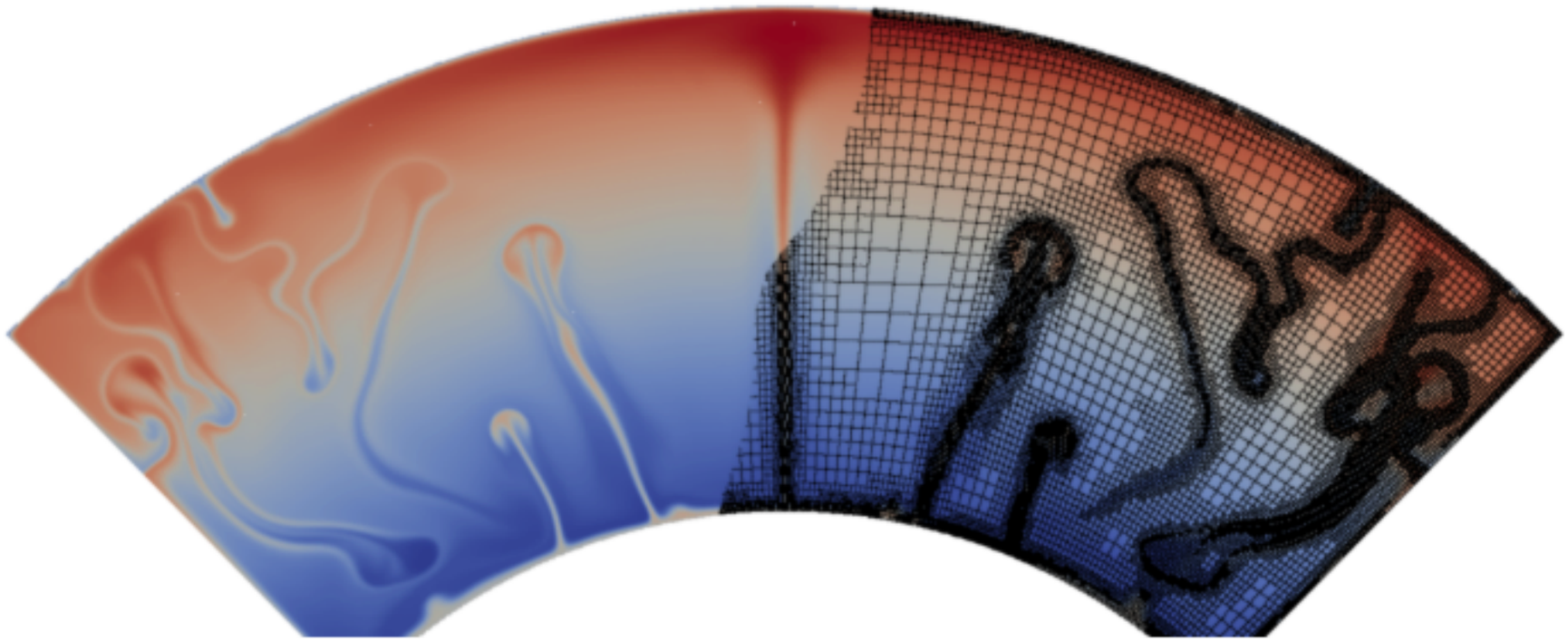
$$h_K \propto \frac{1}{\|u\|_{H^2(K)}}$$

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

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

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:

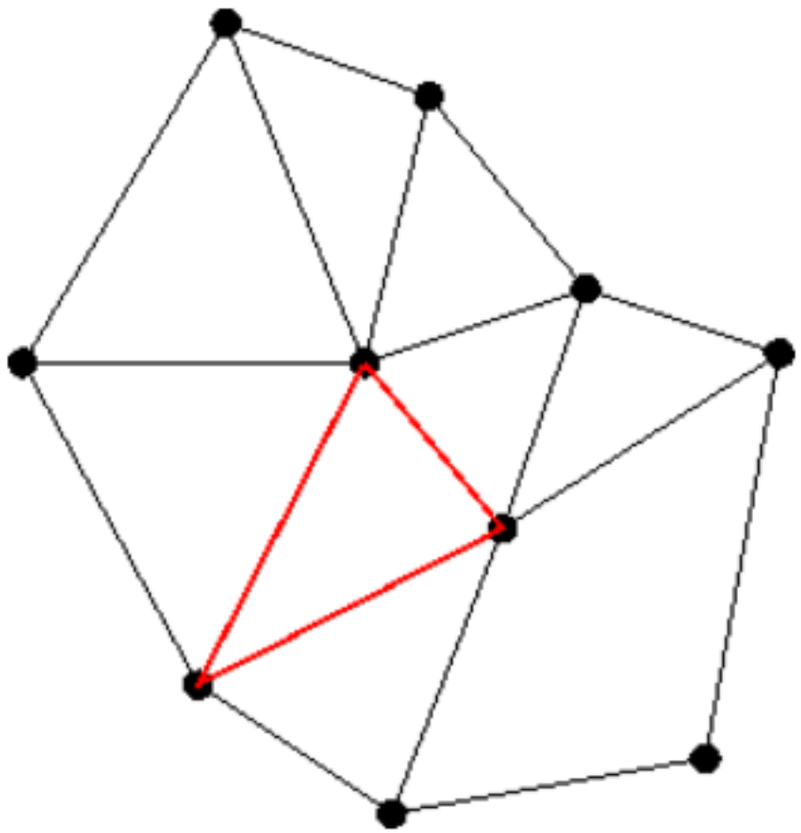
- *De novo* generation of a non-uniform mesh
- Longest edge refinement
- Red-green refinement

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:

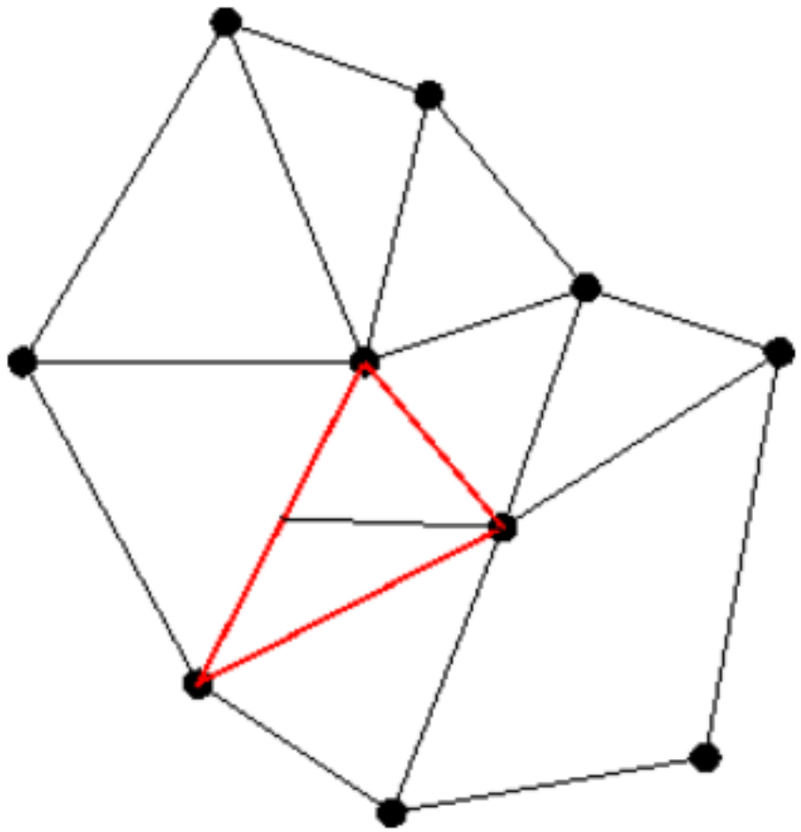


Algorithm:

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

Longest edge refinement

Consider this mesh and a marked cell:

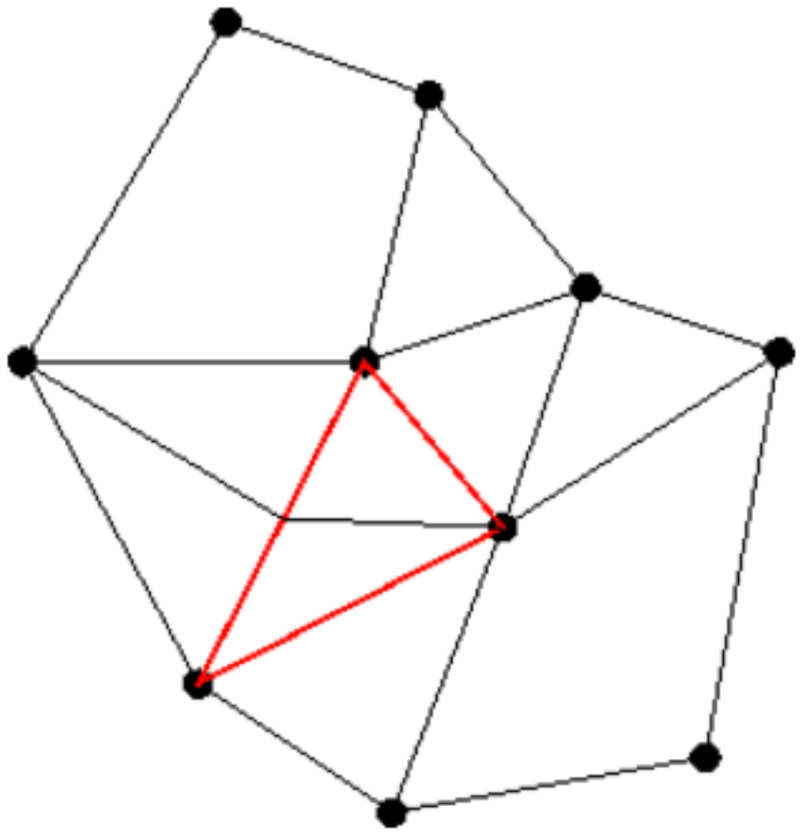


Algorithm:

- 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:

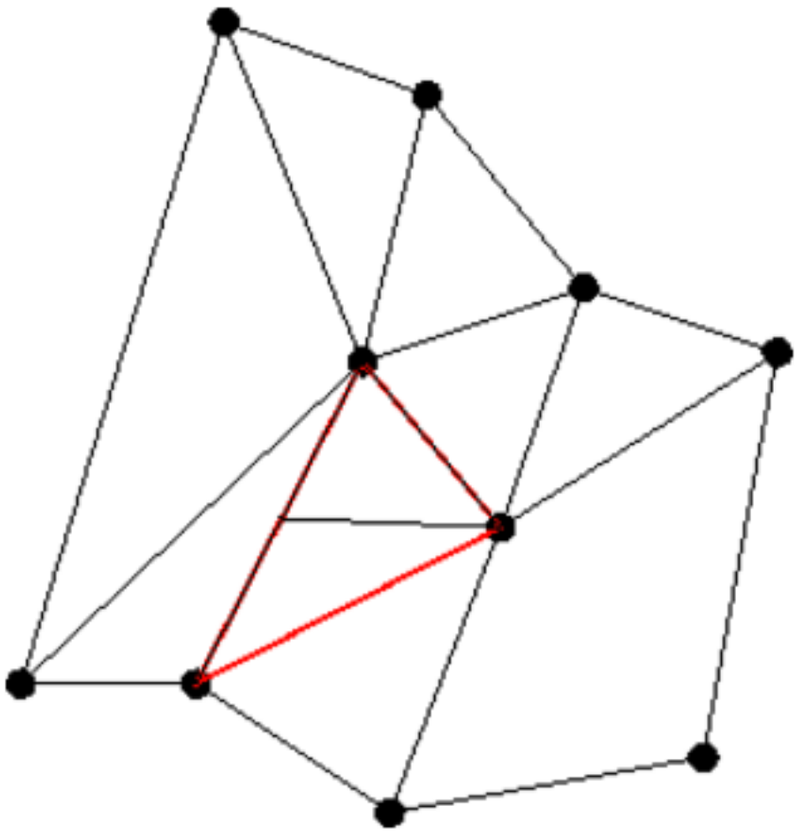


Algorithm:

- 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:

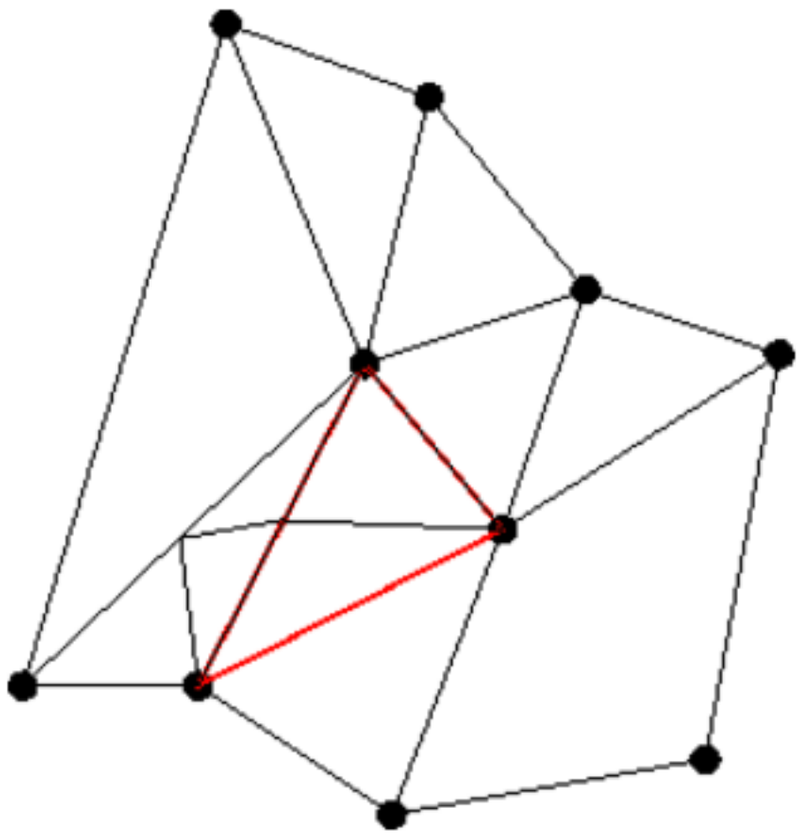


Algorithm:

- 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:

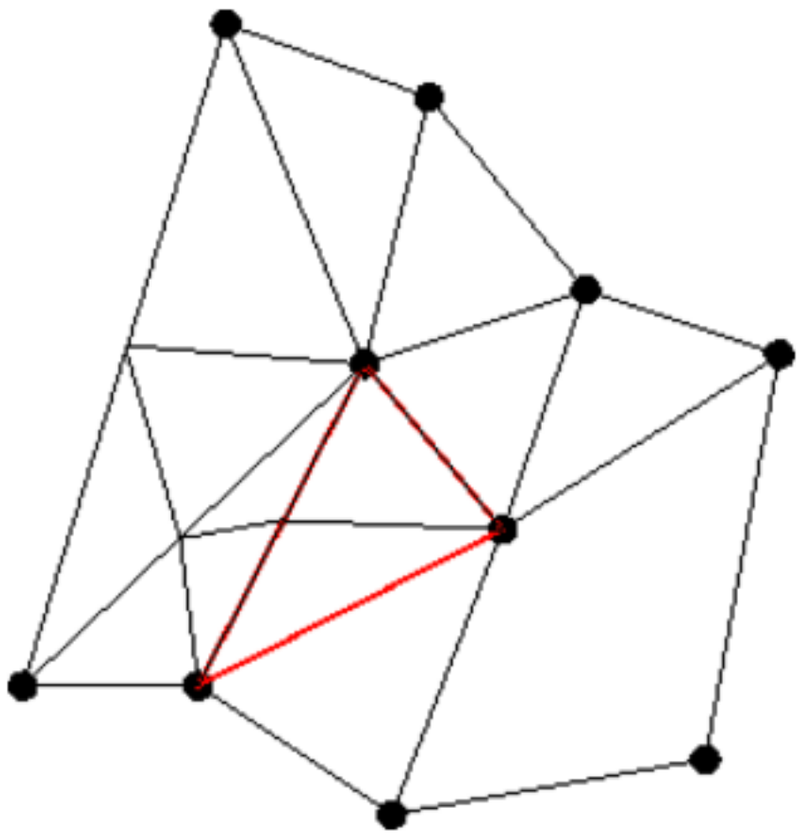


Algorithm:

- 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

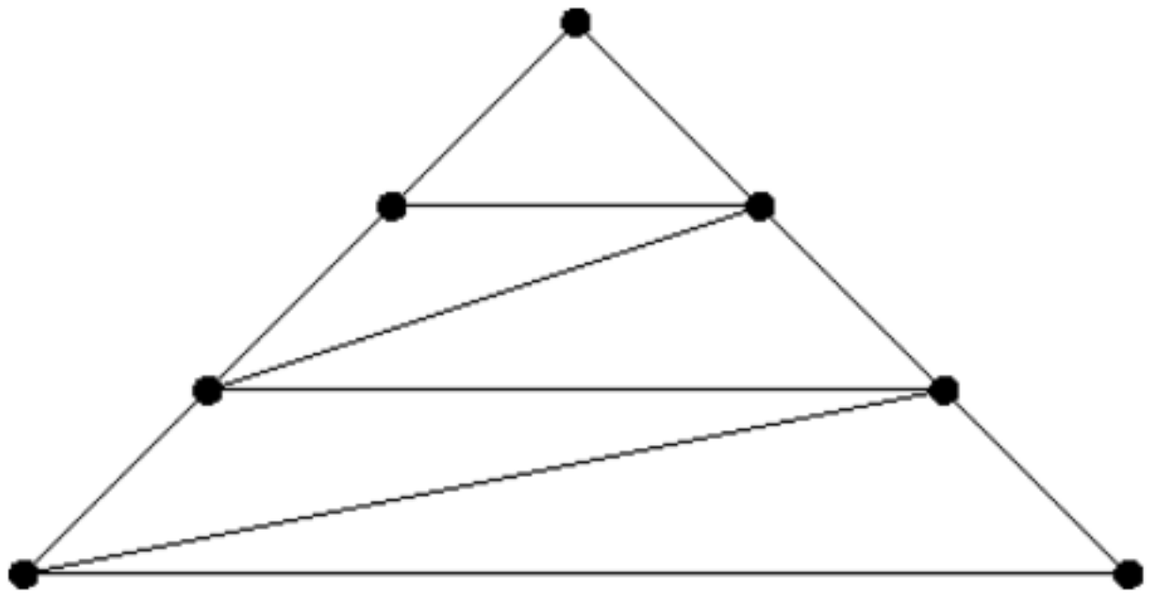
Longest edge refinement

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

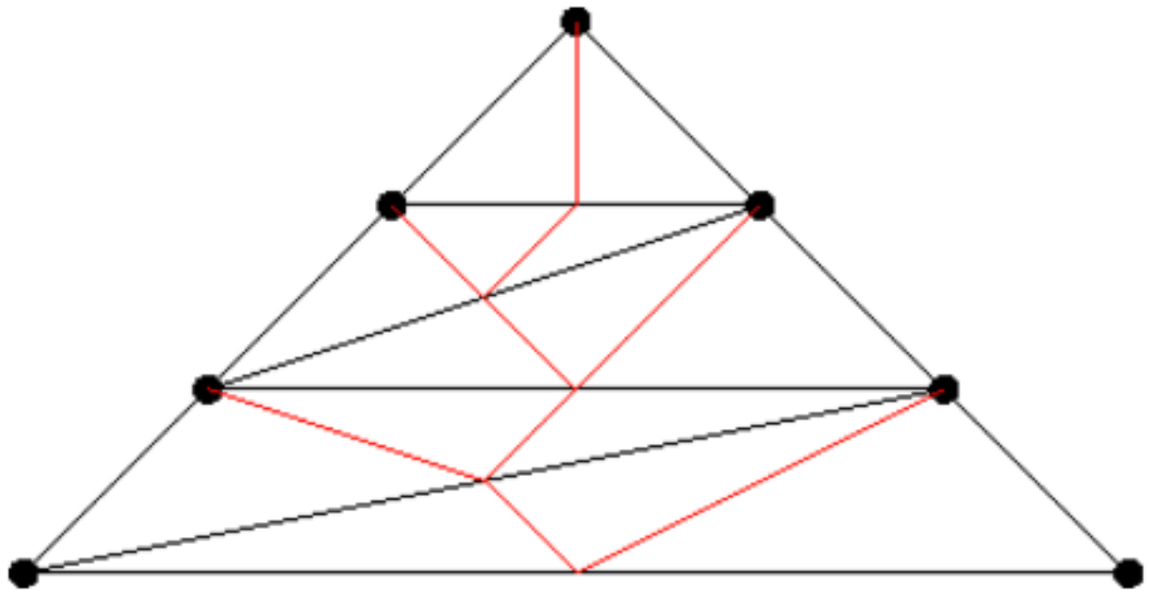
Example of runaway non-local refinement:



Goal: Refine the top-most cell.

Longest edge refinement

Example of runaway non-local refinement:

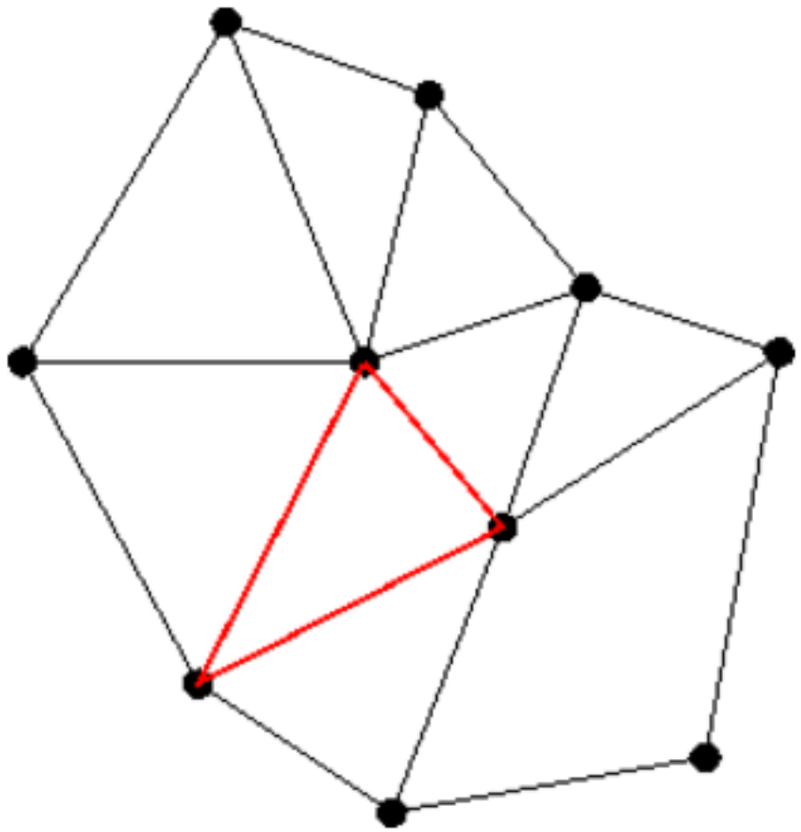


Result: *All* cells are refined!

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:

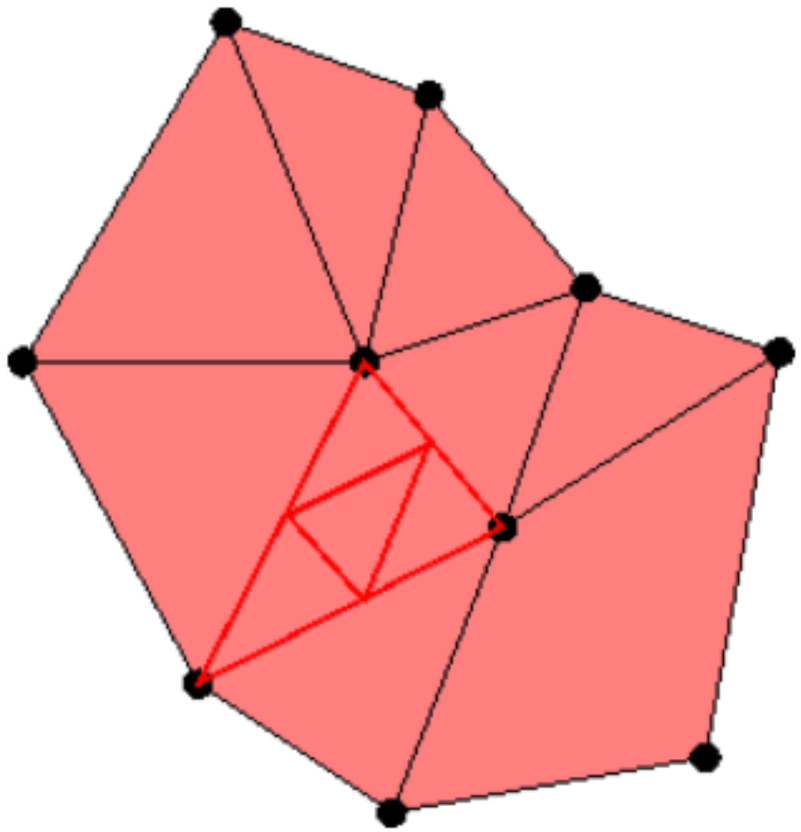


Algorithm:

– All cells start out as “red” cells

Red green refinement

An alternative is as follows:

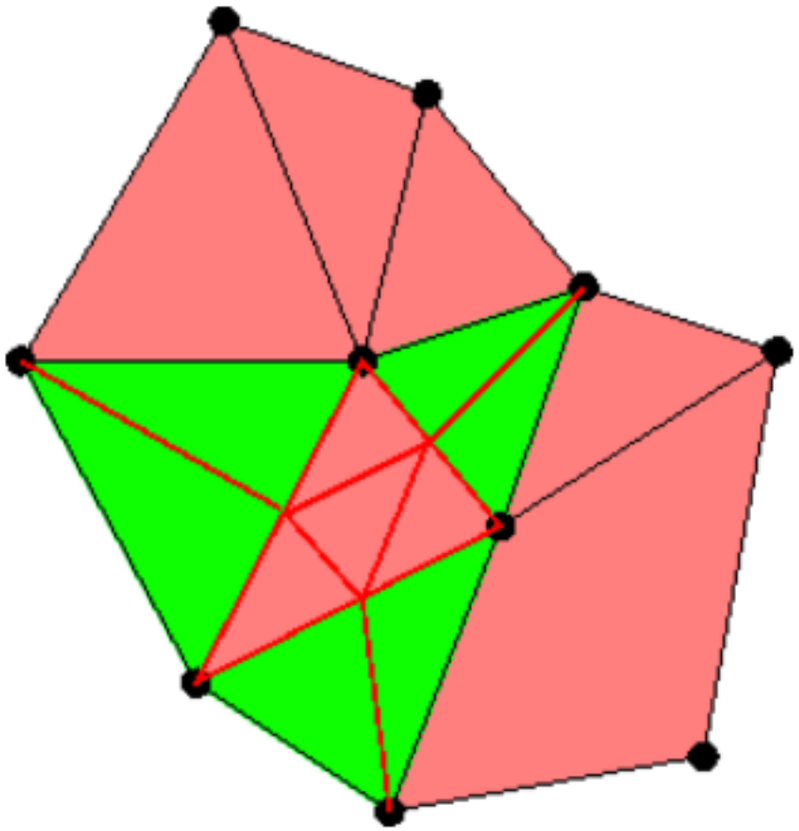


Algorithm:

- 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:

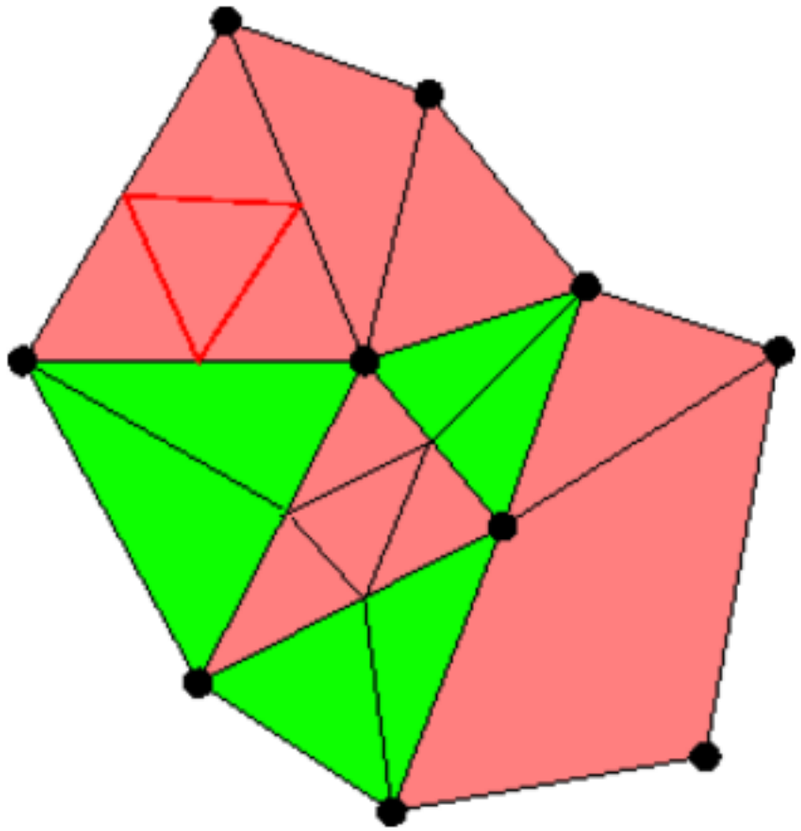


Algorithm:

- 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:

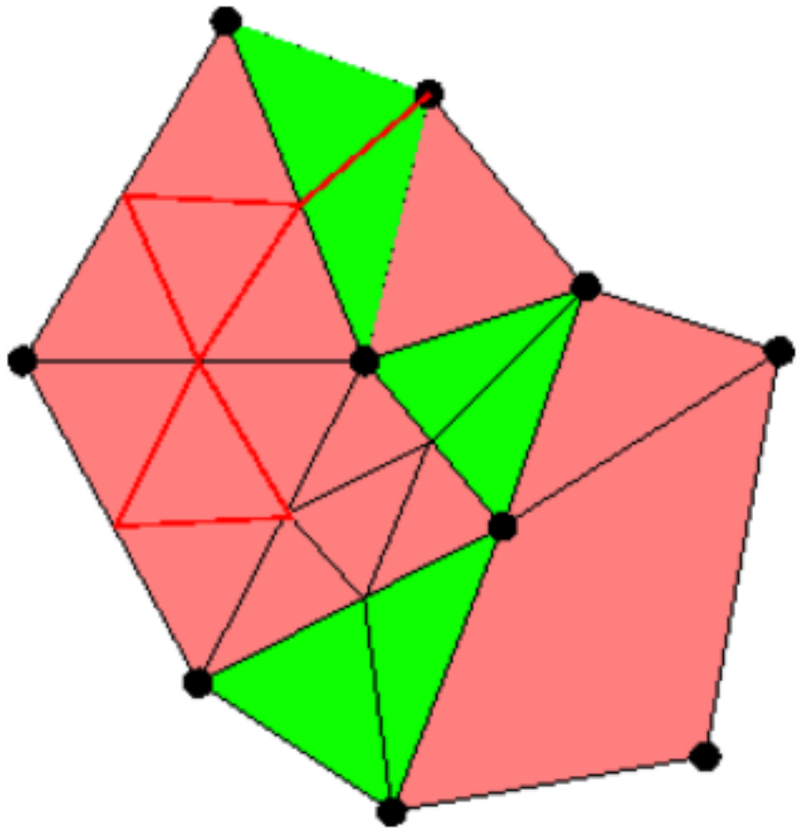


Algorithm:

- 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:

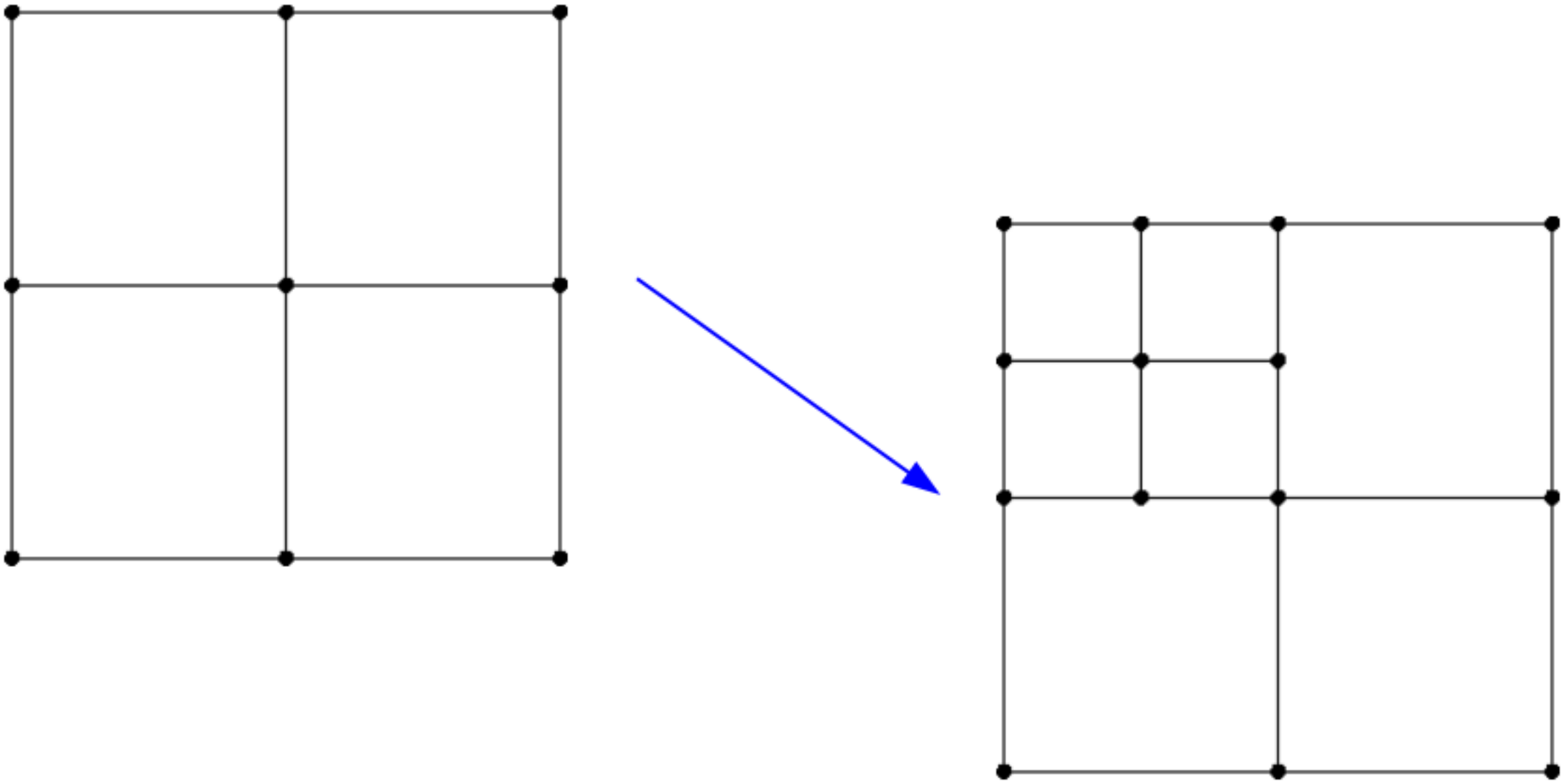
- The algorithm is designed to keep the triangles from degenerating
- Children of "red" cells are congruent to their mother
- No run-away refinement

Cons:

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

Quad/hex refinement

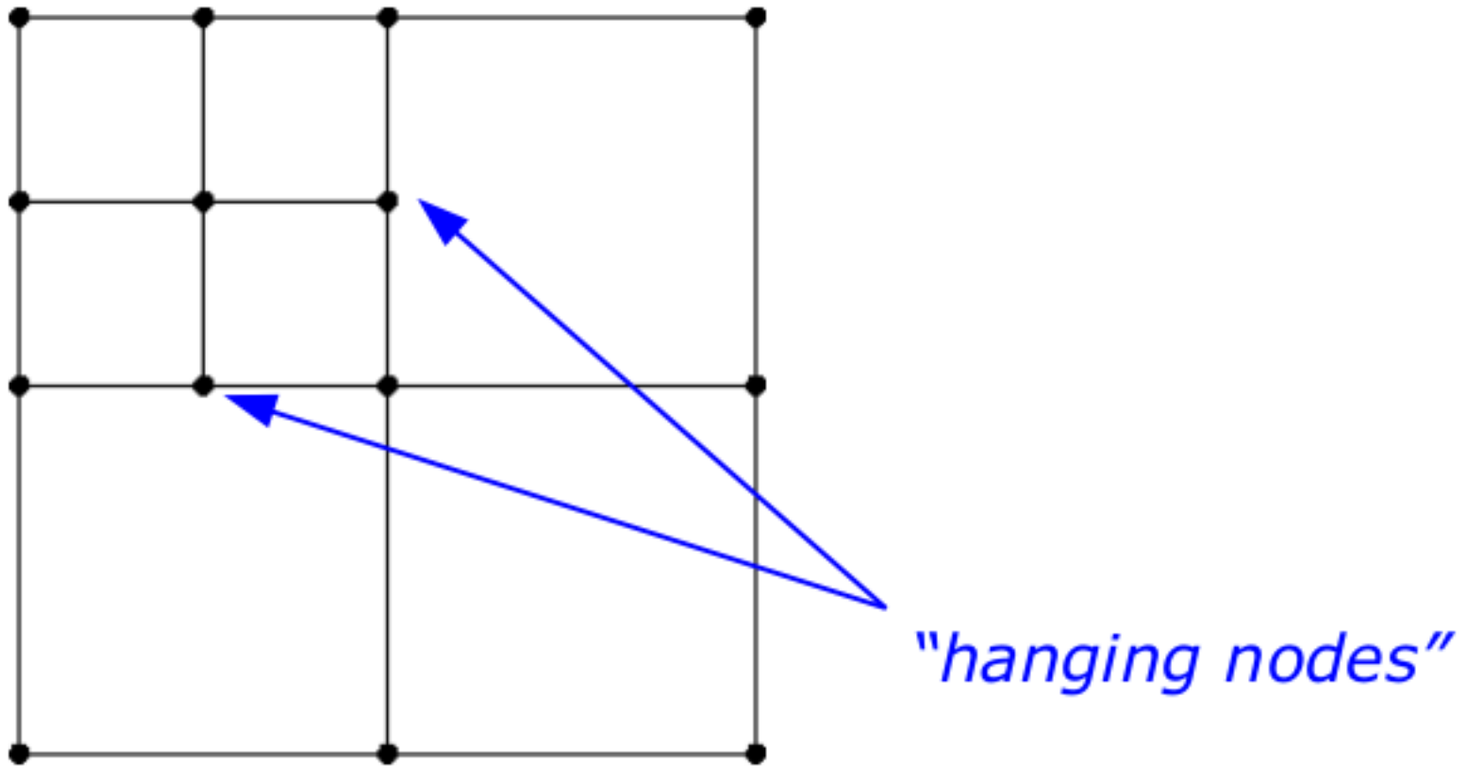
For quads/hexes, the situation is more difficult:



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

Quad/hex refinement

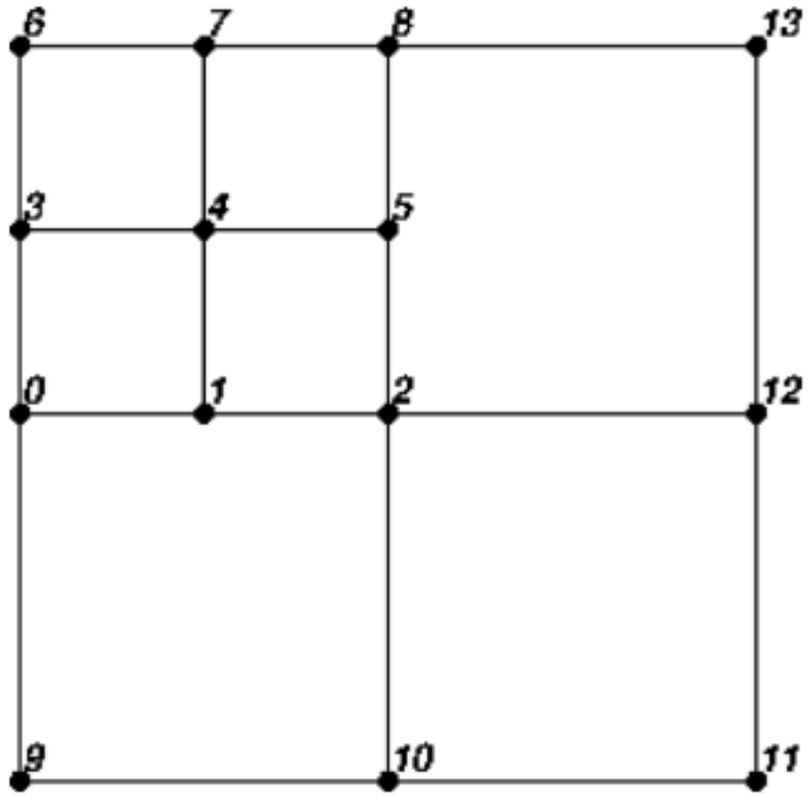
For quads/hexes, the situation is more difficult:



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

Why constraints?

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



The corresponding space has dimension 14.

Local adaptive refinement

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

20 March 2018

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Aims for this module

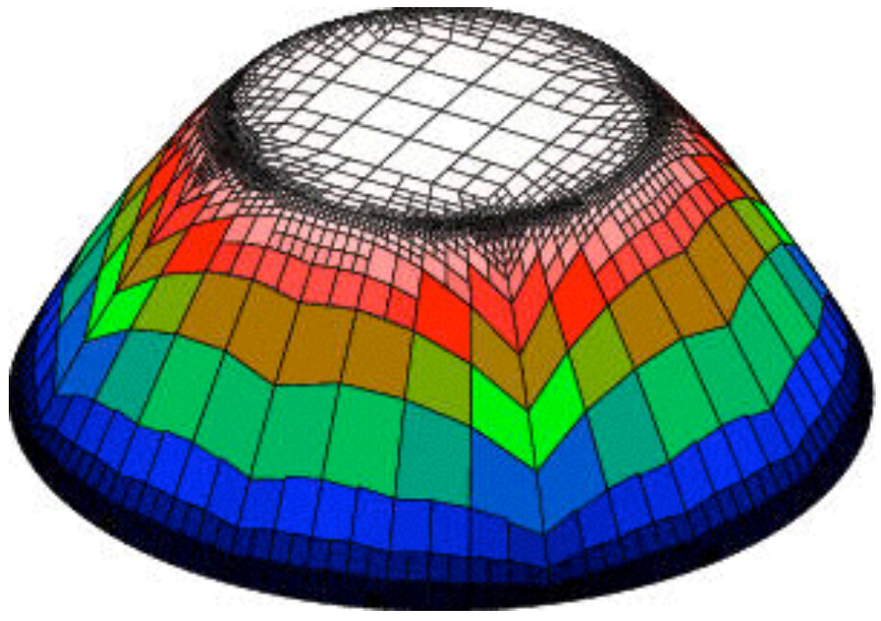
- Implement adaptive mesh refinement
  - Hanging nodes
  - Error-based refinement marking
- Learn about the ConstraintMatrix

Reference material

- Tutorials
  - https://dealii.org/8.5.1/doxygen/deal.II/step_6.html
  - http://www.math.colostate.edu/~bangerth/videos.676.15.html
  - http://www.math.colostate.edu/~bangerth/videos.676.16.html
  - http://www.math.colostate.edu/~bangerth/videos.676.17.html
  - http://www.math.colostate.edu/~bangerth/videos.676.17.25.html
  - http://www.math.colostate.edu/~bangerth/videos.676.17.5.html
  - http://www.math.colostate.edu/~bangerth/videos.676.17.75.html
- Documentation
  - https://dealii.org/8.5.1/doxygen/deal.II/group__constraints.html
  - https://dealii.org/8.5.1/doxygen/deal.II/group__grid.html
  - https://dealii.org/8.5.1/doxygen/deal.II/namespaceGridRefinement.html
  - https://dealii.org/8.5.1/doxygen/deal.II/namespaceDerivativeApproximation.html

Adaptive mesh refinement

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

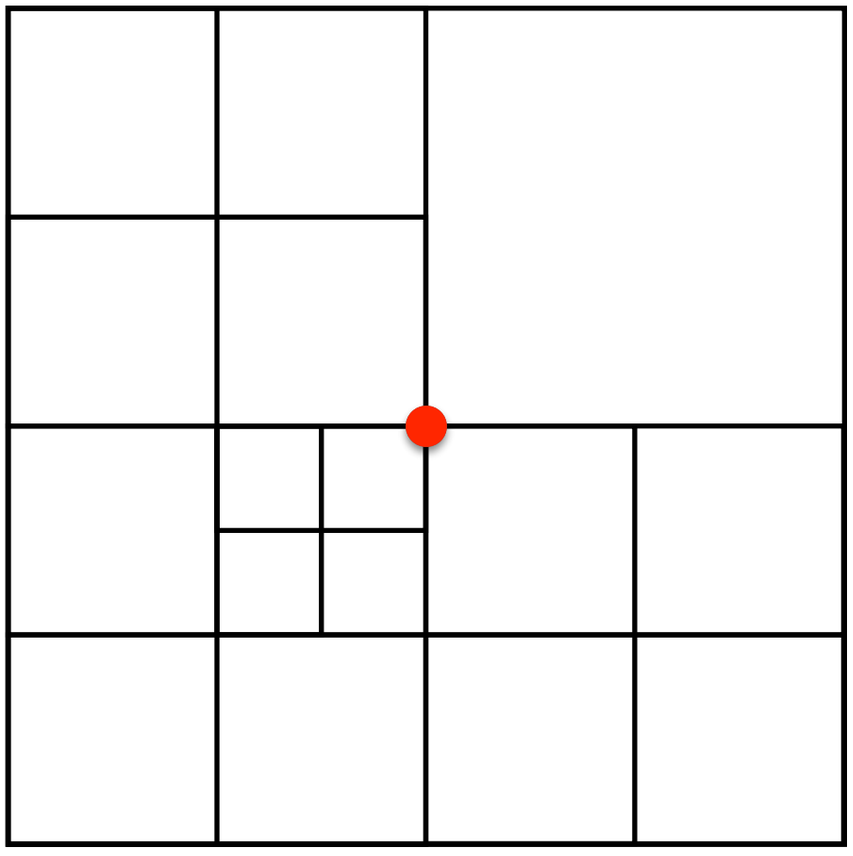


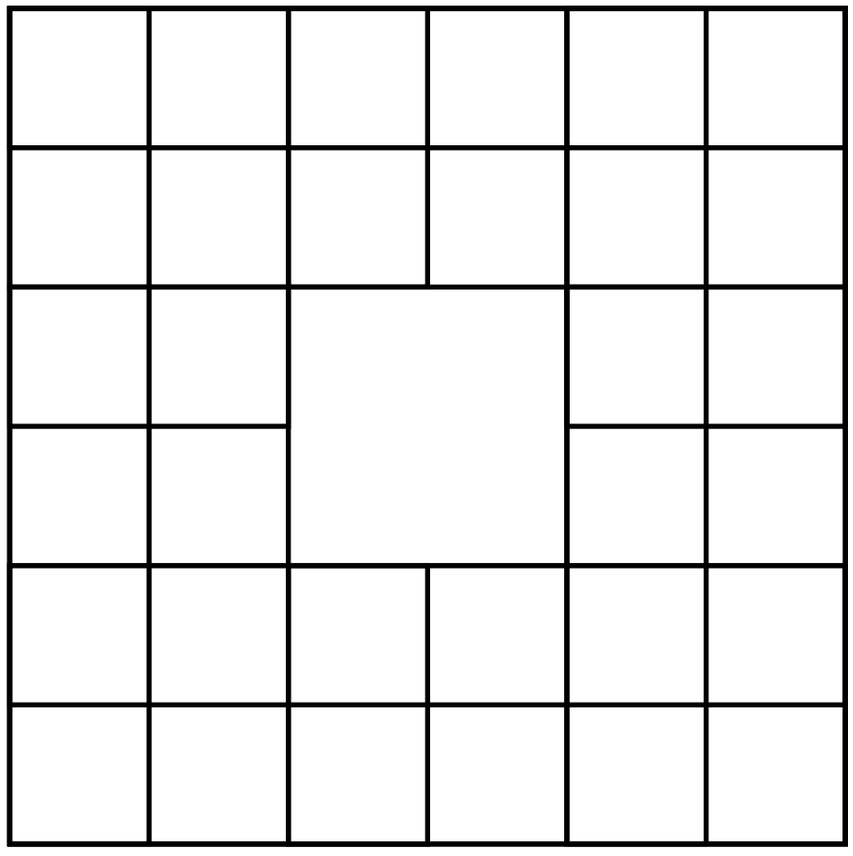
Adaptive mesh refinement

- 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)

Adaptive mesh refinement

- Note: `Triangulation::MeshSmoothing`

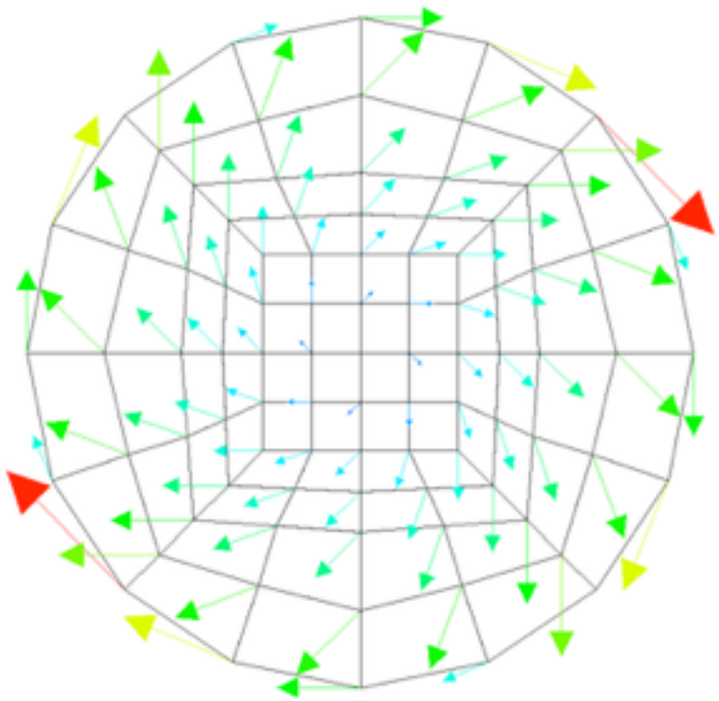




Applying constraints: the ConstraintMatrix class

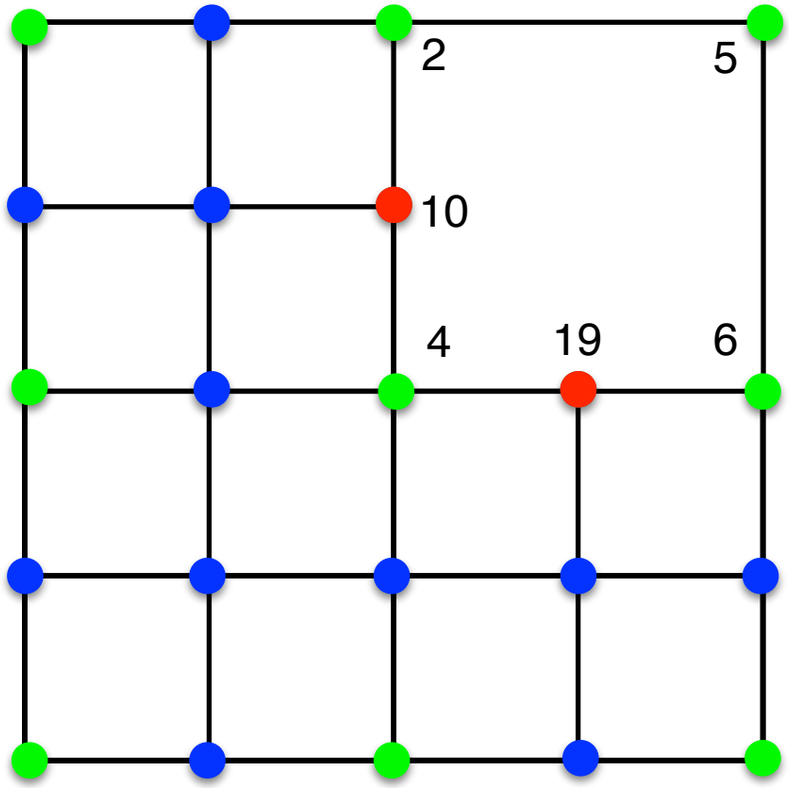
- This class is used for
  - Hanging nodes
  - Dirichlet and periodic constraints
  - Other constraints
- Linear constraints of the the form

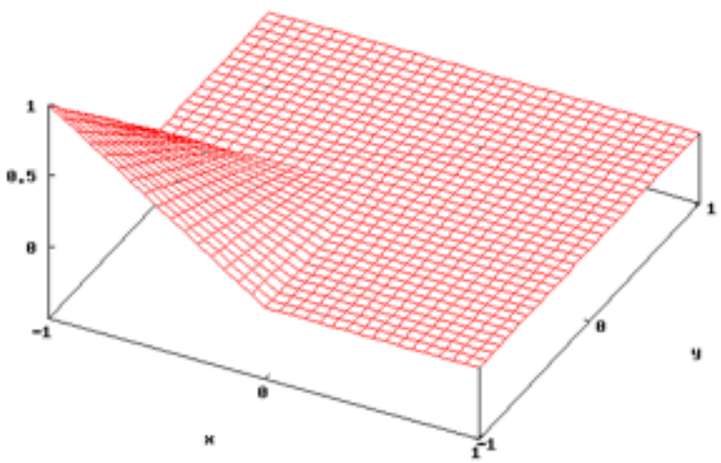
$$x_i = \sum_j \alpha_{ij} x_x + c_j$$

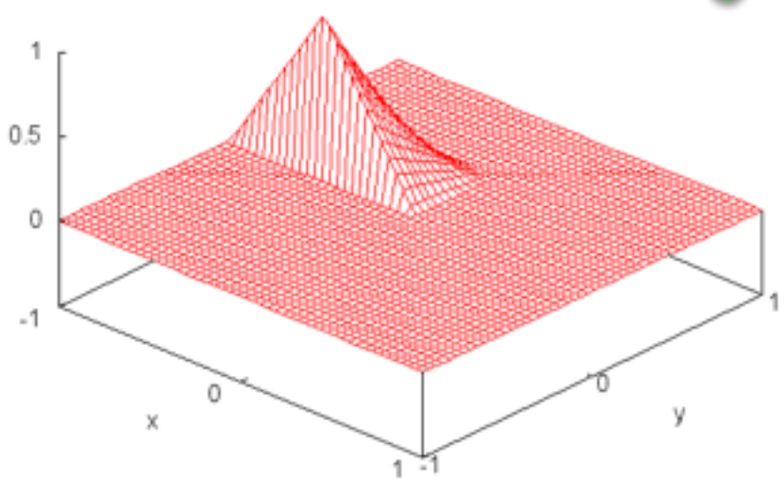


Applying constraints: Hanging node constraints

- Finite element function must be globally continuous
  - Red nodes must have values that are compatible with the adjacent green nodes
  - Function has no jump when traversing from the small cells to the large one at the top right







$$u_{10} = \frac{1}{2}\left[u_2 + u_4\right]$$

$$u_{19} = \frac{1}{2}\left[u_4 + u_6\right]$$

Applying constraints: the ConstraintMatrix 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
- 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

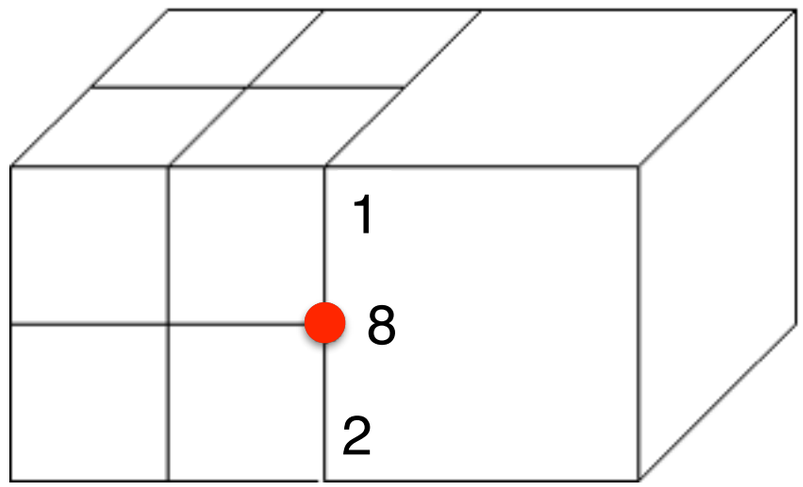
- When writing into a ConstraintMatrix, existing constraints are not overwritten.

- Can merge constraints together:
  `constraints.merge (other_constraints, MergeConflictBehavior::left_object_wins);`

- Which is right?

$$u_8 = \bar{u} \quad \text{or}$$

$$u_8 = \frac{1}{2}\left[u_1 + u_2\right]$$



- Beware of introducing constraint cycles

$$u_1 = u_2 \; ; \; u_2 = u_3 \; ; \; u_3 = u_1$$

Shared memory parallelisation

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

20 March 2018

Aims for this module

- Identify parts / blocks of code that are (easily) parallelisable

- Learn how to parallelise using
  - ThreadGroup (Posix threads)
  - Workstream (Threaded building blocks)

Reference material

- Tutorials
  - https://dealii.org/8.5.1/doxygen/deal.II/step_9.html
  - https://dealii.org/8.5.1/doxygen/deal.II/step_13.html
  - http://www.math.colostate.edu/~bangerth/videos.676.39.html
  - http://www.math.colostate.edu/~bangerth/videos.676.40.html
- Documentation:
  - https://dealii.org/8.5.1/doxygen/deal.II/group__threads.html
  - https://www.dealii.org/8.5.1/doxygen/deal.II/namespaceWorkStream.html
  - https://dealii.org/8.5.1/doxygen/deal.II/namespaceparallel.html

Identifying parallelisable code

- Consider this example:

```
template <int dim>
void MyProblem<dim>::setup_system (){
  dof_handler.distribute_dofs();
  DoFTools::make_hanging_node_constraints (…);    // 1
  DoFTools::make_sparsity_pattern (…);            // 2
  VectorTools::interpolate_boundary_values (…);   // 3
…
}
```

- Operations (1,2,3) are independent of one another
  - Could be reordered without consequence

Identifying parallelisable code

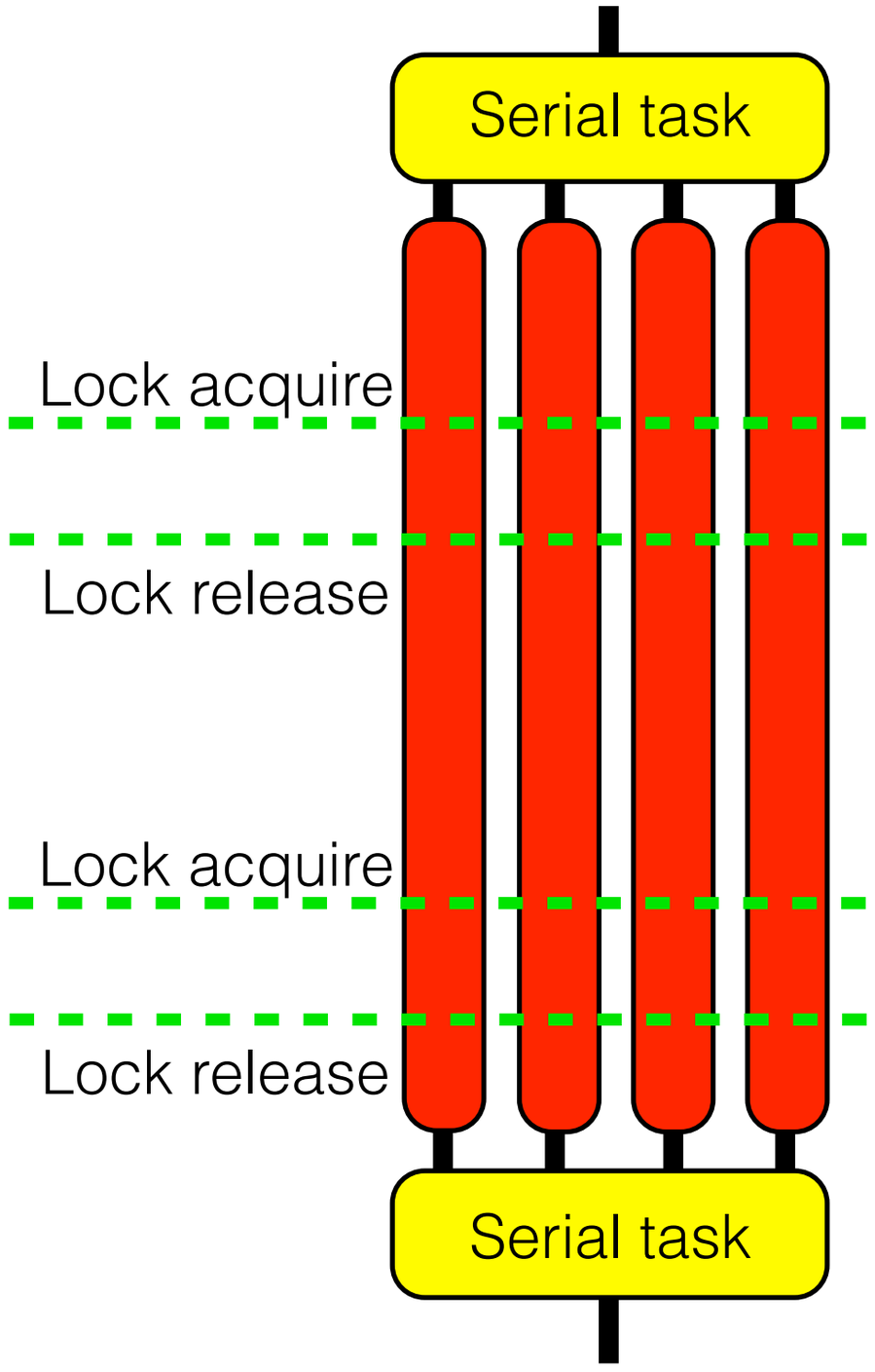
- “Embarrassingly parallelisable tasks”

```
template <int dim>
void MyProblem<dim>::assemble_system (){
…
for (cell=dof_handler.begin_active(); …)
{
  fe_values.reinit (cell);
  ...assemble local contribution...
  ...copy local contribution into global matrix/rhs vector...
}}
```

- Many more cells than machine cores
- Computations are mutually independent
  - Accumulation into global system is not

Independent threaded tasks: Option 1

- Code divergence with / without barriers (global / in-thread locks)
- Best used for small number of completely independent tasks
- Inside each thread: Shared data
  - Reading is a safe operation!
  - Use locks to allow data writing
    - Convergence point for threads (bottleneck)
    - Potential for deadlocks



Creating independent threaded tasks: the Thread class

```
template <int dim>
void MyProblem<dim>::setup_system (){
  dof_handler.distribute_dofs();

  Threads::Thread<void> thread1, thread2, thread3;

  thread1 = Threads::new_thread (&DoFTools::make_hanging_node_constraints,…);
  thread2 = Threads::new_thread (&DoFTools::make_sparsity_pattern, …);
  thread3 = Threads::new_thread (&VectorTools::interpolate_boundary_values,,…);

  thread1.join();     // and same for thread2, thread3
  …
}
```

- The call to join() is a blocking call
  - Waits to the thread to finish before continuing

Creating independent threaded tasks: the ThreadGroup class

```
void MyProblem<dim>::assemble_on_one_cell (cell_iterator &cell) {…}

void MyProblem<dim>::assemble_system () {
  Threads::ThreadGroup<void> threads;

  for (cell=dof_handler.begin_active(); …)
    threads += Threads::new_thread (
      &MyProblem<dim>::assemble_on_one_cell,
      this, cell);

  threads.join_all ();
}
```

- Why is this inefficient?
- How do we prevent data races?

Creating independent threaded tasks: Ranged based assembly

- Less threads created = more efficient

```
void MyProblem<dim>::assemble_on_cell_range (
  cell_iterator &range_begin,
  cell_iterator &range_end) {…};

void MyProblem<dim>::assemble_system () {
  Threads::ThreadGroup<void> threads;

  std::vector<std::pair<cell_iterator, cell_iterator> >
    sub_ranges = Threads::split_range (
      dof_handler.begin_active(),
      dof_handler.end(),
      n_virtual_cores);

  for (t=0; t<n_virtual_cores; ++t)
    threads += Threads::new_thread (
      &MyProblem<dim>::assemble_on_cell_range,
      this,
      sub_ranges[t].first,
      sub_ranges[t].second);

  threads.join_all ();}
}
```

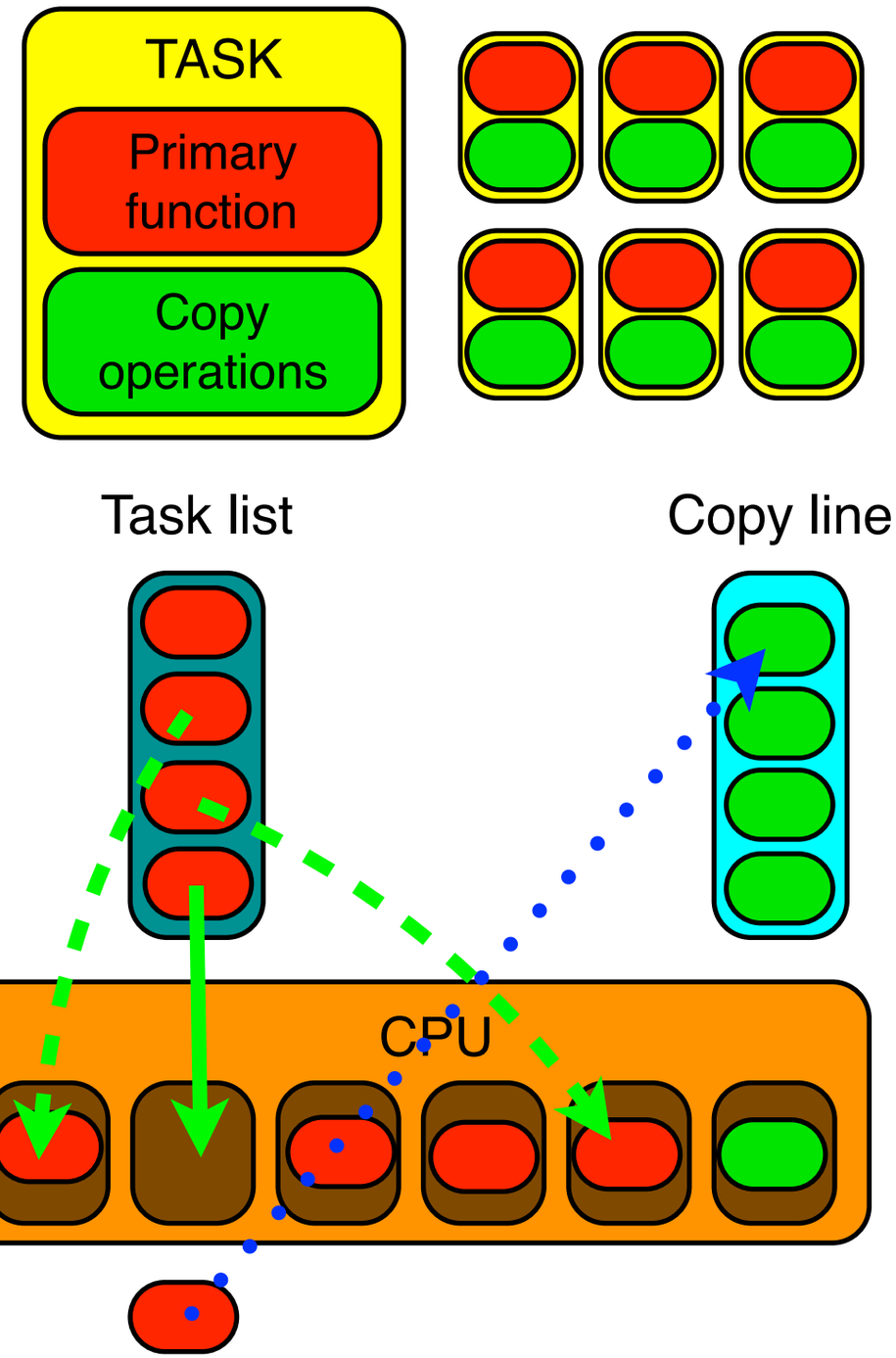
Independent threaded tasks

- How do we prevent data races?

```
void MyProblem<dim>::assemble_on_one_cell (cell_iterator &cell) {

  static Threads::Mutex mutex;

  mutex.acquire ();
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      system_matrix.add (dof_indices[i], dof_indices[j],
                         cell_matrix(i,j));
  ...same for rhs...
  mutex.release ();
}
```

Creating independent threaded tasks: the WorkStream class

- Task-based threading
  - Continuous use of free CPU cores
  - Create a list of tasks
  - When core free, use it to perform next task
    - Expensive operations continually executed
  - Perform blocking tasks independently
    - Data copied to shared objects serially
  - Optimisations:
    - “Automatic” load balancing
    - Overhead reduction: Works on data chunks



Creating independent threaded tasks: parallelisation of (per-cell) assembly

```
template <int dim>
void MyClass<dim>::assemble_on_one_cell (
   const typename DoFHandler<dim>::active_cell_iterator &cell)
{
  FEValues<dim> fe_values (...);

  FullMatrix<double> cell_matrix (...);
  Vector<double>     cell_rhs (...);
  std::vector<double> rhs_values (...);

  rhs_function.value_list (...)

  // assemble local contributions
  fe_values.reinit (cell);
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      for (unsigned int q=0; q<n_points; ++q)
        cell_matrix(i,j) += ...;
  ...same for cell_rhs...

  // now copy results into global system
  std::vector<unsigned int> dof_indices (...);
  cell->get_dof_indices (dof_indices);
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      system_matrix.add (...);
  ...same for rhs...
  // or constraints.distribute_local_to_global (...);
}
```

Expensive constructor call

Repeated memory allocation

Independent tasks

Serial operation

Threading using WorkStream: the StratchData class

- Assistant struct / class
- Contains reused data structures
  - FEValues objects
  - Helper vectors and storage containers
  - Precomputed data
- Needs a constructor and a copy constructor
  - Some objects must be manually reconstructed
  - We create one initial instance of the class
  - TBB duplicates as required (queue_length)

```
struct ScratchData {
  std::vector<double>        rhs_values;
  FEValues<dim>              fe_values;

  ScratchData (
    const FiniteElement<dim> &fe,
    const Quadrature<dim>    &quadrature,
    const UpdateFlags         update_flags)
    : rhs_values (quadrature.size()),
      fe_values (fe, quadrature, update_flags)
    {}

   ScratchData (const ScratchData &rhs)
   : rhs_values (rhs.rhs_values),
     fe_values (rhs.fe_values.get_fe(),
                rhs.fe_values.get_quadrature(),
                rhs.fe_values.get_update_flags())
    {}
}
```

Threading using WorkStream: the PerTaskData class

- Contains data structures required for serial operations
  - Multiple copies made (queue_length*chunk_size)
  - Must be “self-contained”
- Used in two places
  - Threaded function
    - Bound to an instance of the threaded function
    - Used as a “data-in” object
  - Serial function
    - A used instance is passed to this function
    - Used as a “data-out” object

```
struct PerTaskData {
  FullMatrix<double>          cell_matrix;
  Vector<double>              cell_rhs;
  std::vector<unsigned int> dof_indices;

  PerTaskData (const FiniteElement<dim> &fe)
  : cell_matrix (fe.dofs_per_cell,
                 fe.dofs_per_cell),
    cell_rhs (fe.dofs_per_cell),
    dof_indices (fe.dofs_per_cell)
  {}
}
```

Threading using WorkStream: Revised assembly

```
template <int dim>
void MyClass<dim>::assemble_on_one_cell (
   const typename DoFHandler<dim>::active_cell_iterator &cell,
   ScratchData &scratch,
   PerTaskData &data)
{
  // reinitialise data
  scratch.fe_values.reinit (cell);
  rhs_function.value_list (scratch.fe_values.get_quadrature_points,
                           scratch.rhs_values);
  ...
  data.cell_matrix = 0;
  data.cell_rhs    = 0;

  // assemble local contributions
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      for (unsigned int q=0; q<fe_values.n_quadrature_points; ++q)
        data.cell_matrix(i,j) += ...;
  ...
}
```

- Now use objects contained within ScratchData and PerTaskData structs

Threading using WorkStream: Serial copy operation

```
template <int dim>
void MyClass<dim>::copy_local_to_global (const PerTaskData &data)
{
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
      system_matrix.add (data.dof_indices[i], data.dof_indices[j],
                         data.cell_matrix(i,j));
  ...same for rhs…
  // or constraints.distribute_local_to_global (...);
}
```

- Uses writes “fixed” data in PerTaskData to single class object system_matrix (and whatever else)

- Has to be a serially performed operation

Threading (not) using WorkStream: Manual assembly using these data structures

```
ScratchData scratch_data (…);
PerTaskData per_task_data (…);

DoFHandler<deal_II_dimension>::active_cell_iterator
   cell = dof_handler.begin_active(),
   endc = dof_handler.end();
   for (; cell != endc; ++cell)
{
   assemble_system_one_cell(cell,
                            scratch_data,
                            per_task_data);
   copy_local_to_global(per_task_data);
}
```

- This performs the same serial assembly as we had before
  - More efficient though (use of ScratchData)

Threading using WorkStream

```
ScratchData scratch_data (…);
PerTaskData per_task_data (…);

WorkStream::run ( dof_handler.begin_active(),
                  dof_handler.end(),
                  *this,
                  &MyClass::assemble_system_one_cell,
                  &MyClass::copy_local_to_global,
                  scratch_data,
                  per_task_data );
```

- Execute function in threaded manner
- Only operates on functions with a specific prototype
  - Theadable function:
    void function_name(cell, scratch, per_task_data)
  - Serial function:
    void function_name(per_task_data)

Threading using WorkStream

```
WorkStream::run ( dof_handler.begin_active(),
                  dof_handler.end(),
                  std::bind(&Solid::assemble_residual_one_cell,
                            this,
                            _1,  // cell
                            _2,  // scratch
                            _3), // data
                  std::bind(&Solid::copy_local_to_global_residual,
                            this,
                            _1, // data
                           &in_vector),
                  scratch_data,
                  per_task_data_residual);
```

- copy_local_to_global_F function prototype:
  void function (per_task_data, vector)

- std::bind only binds memory addresses

  - Will make copies of objects not sent in via memory address

  - Need to send in pointers if wish to work on an existing object

  - “std::_1, _2, _3” are placeholders for expected data

MPI parallelisation: Part 1

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

21 March 2018

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

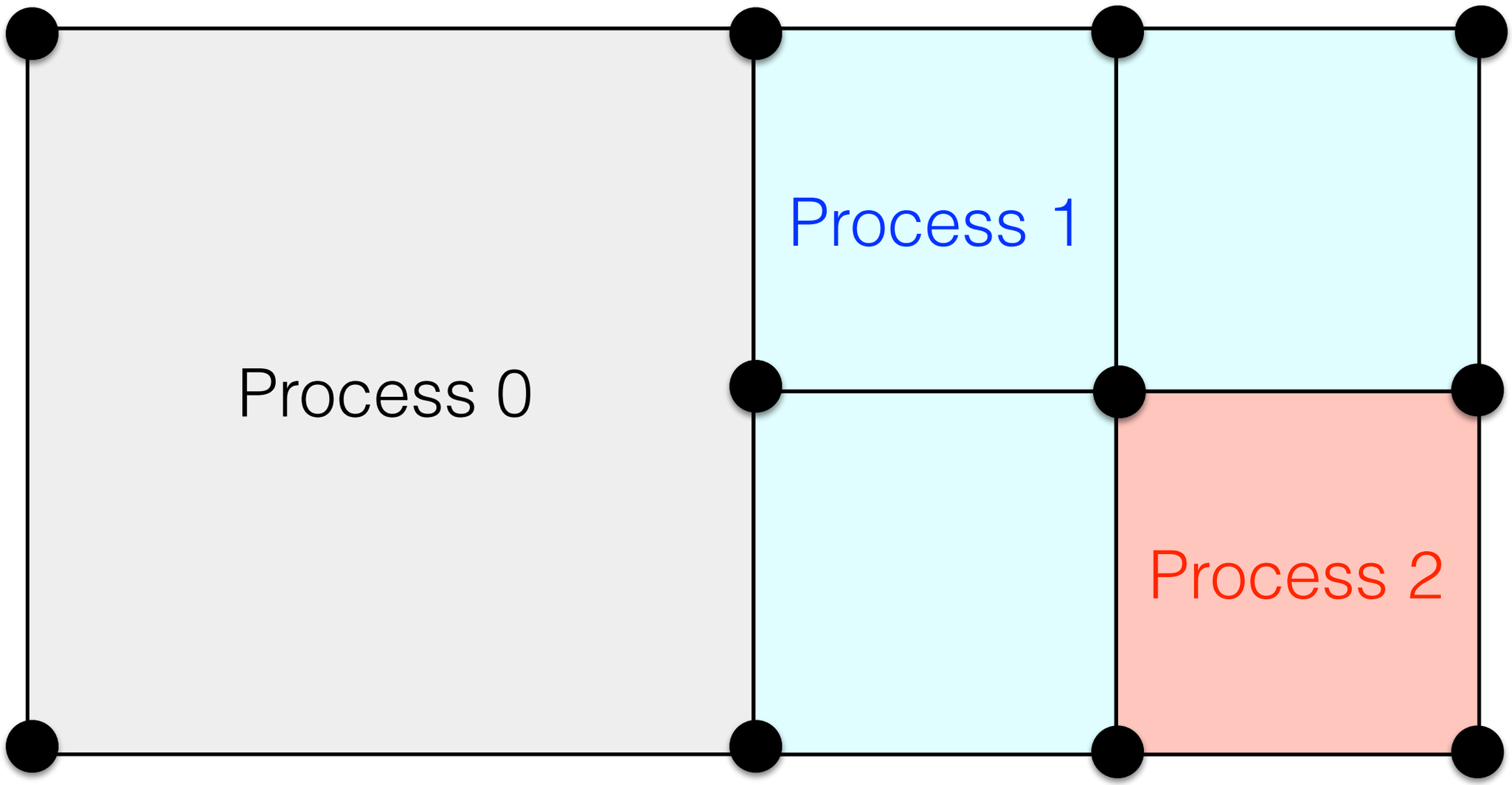
Aims for this module

- First introduction into parallel computing with deal.II
- Parallel distribution of degrees-of-freedom
  - Ownership concepts
- Setup data structures for parallel processing
- Assembly in parallel
- Synchronisation of distributed data
- Visualisation of distributed solutions

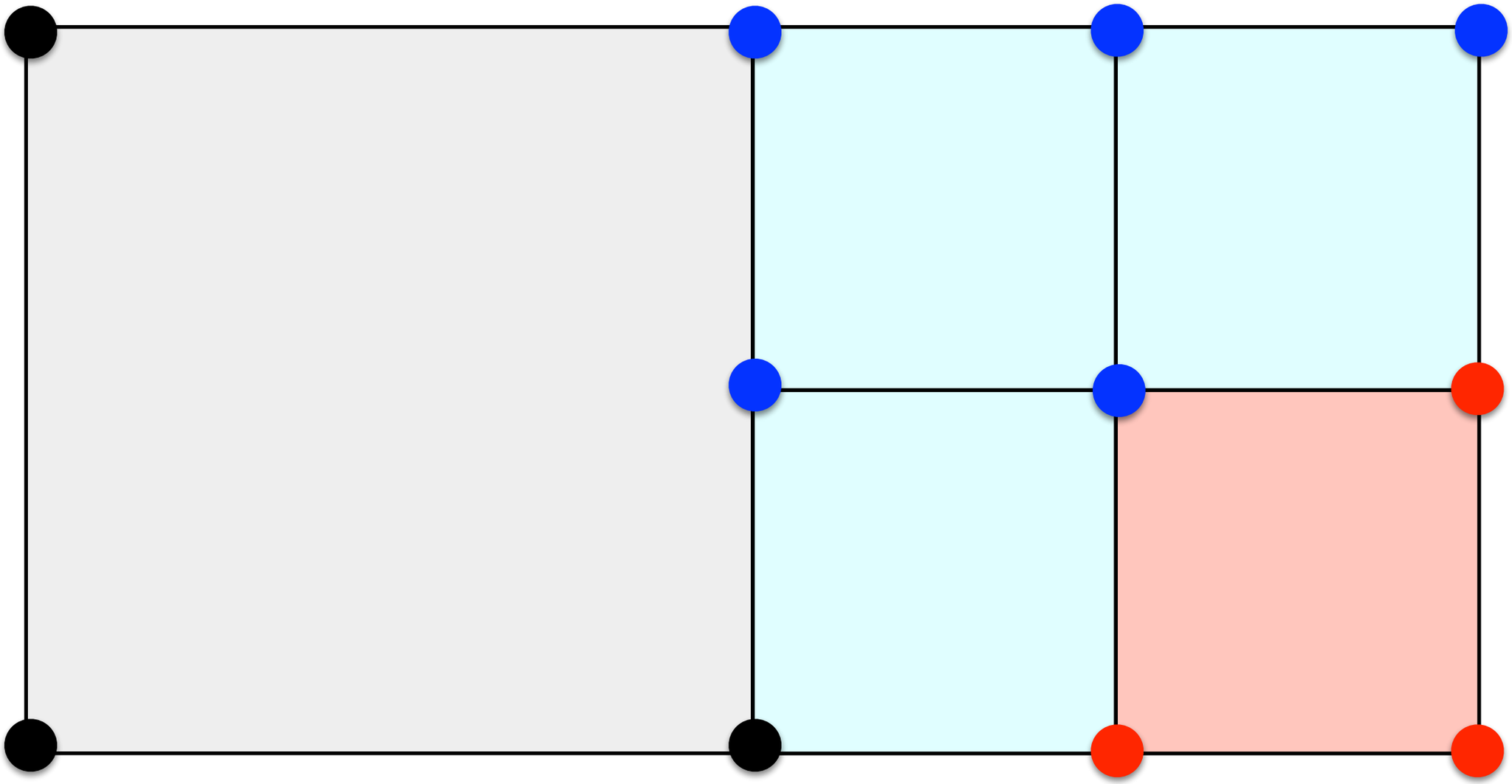
Reference material

- Tutorials
  - https://dealii.org/8.5.1/doxygen/deal.II/step_17.html
  - https://dealii.org/8.5.1/doxygen/deal.II/step_18.html
  - http://www.math.colostate.edu/~bangerth/videos.676.39.html
  - http://www.math.colostate.edu/~bangerth/videos.676.41.html
  - http://www.math.colostate.edu/~bangerth/videos.676.41.25.html
  - http://www.math.colostate.edu/~bangerth/videos.676.41.5.html
  - http://www.math.colostate.edu/~bangerth/videos.676.41.75.html
- Documentation:
  - https://www.dealii.org/developer/doxygen/deal.II/group__distributed.html

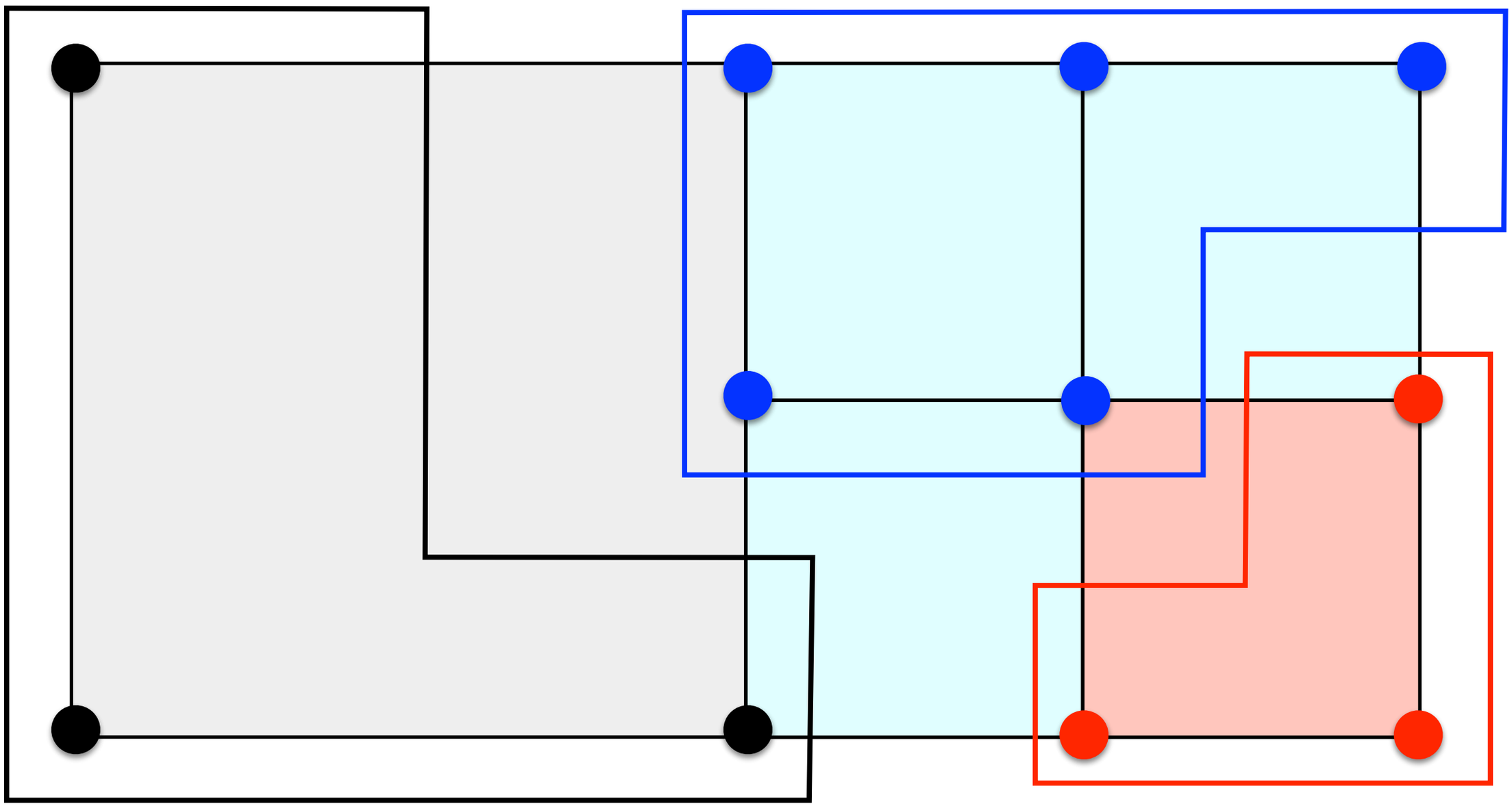
Distribution of degrees-of-freedom: Colourisation of cell ownership via graph partitioner



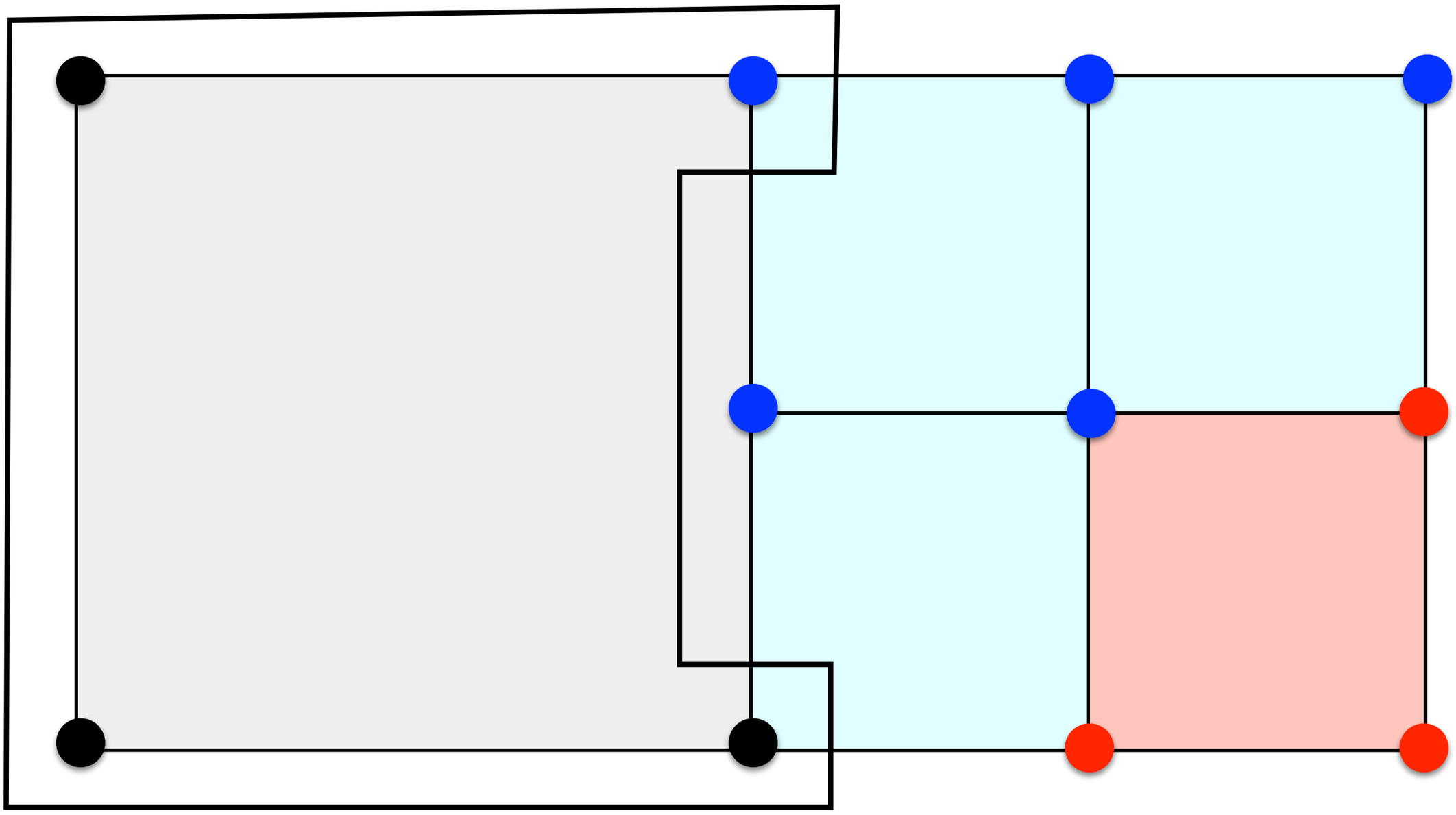
Distribution of degrees-of-freedom: Colourisation of DoFs via graph partitioner



Ownership of distributed DoFs: Locally owned degrees-of-freedom

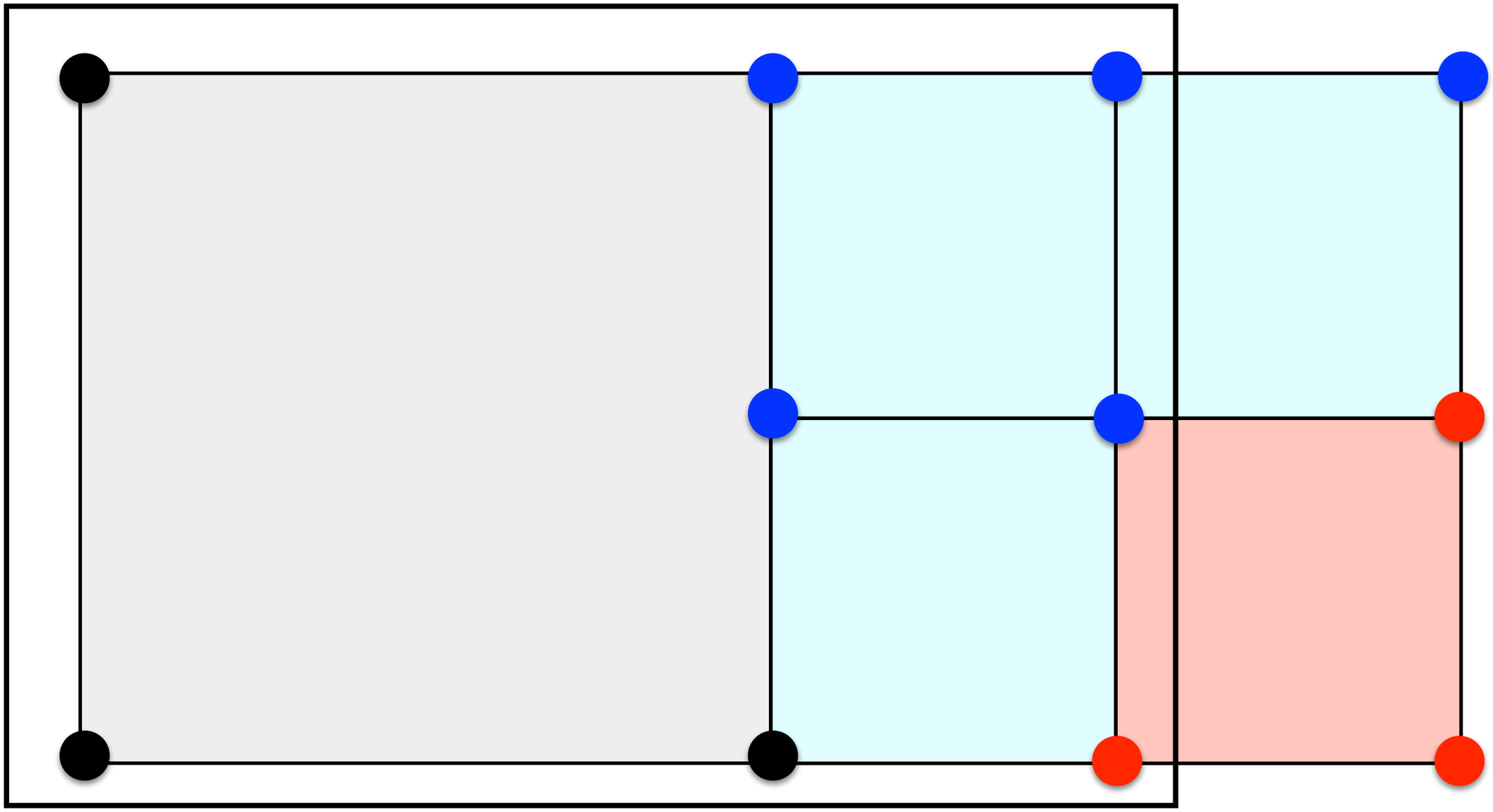


Distribution of DoFs:
Locally relevant degrees-of-freedom (process 0)



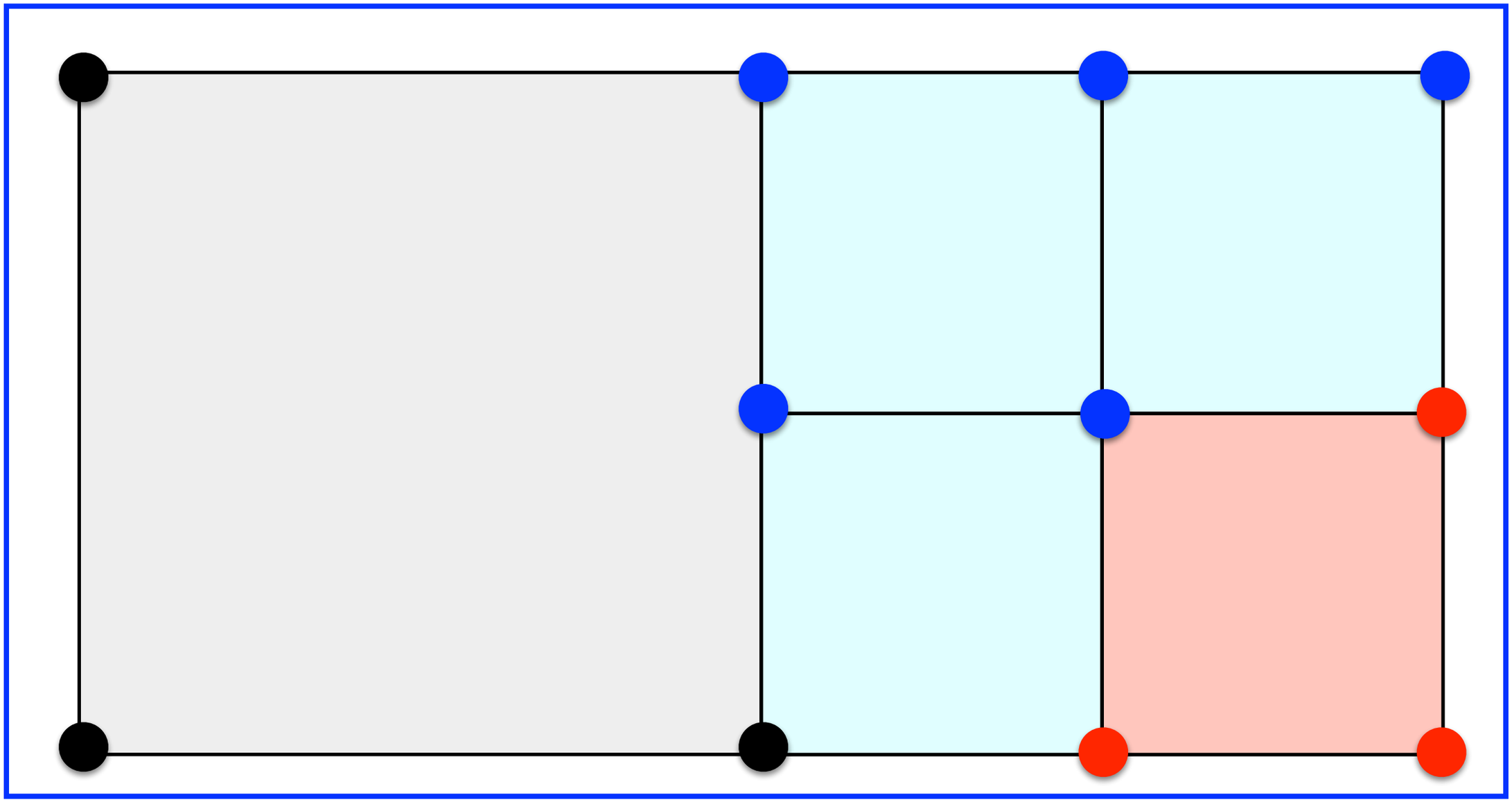
- Required for assembly, data output

Distribution of DoFs:
Locally relevant degrees-of-freedom (process 0)



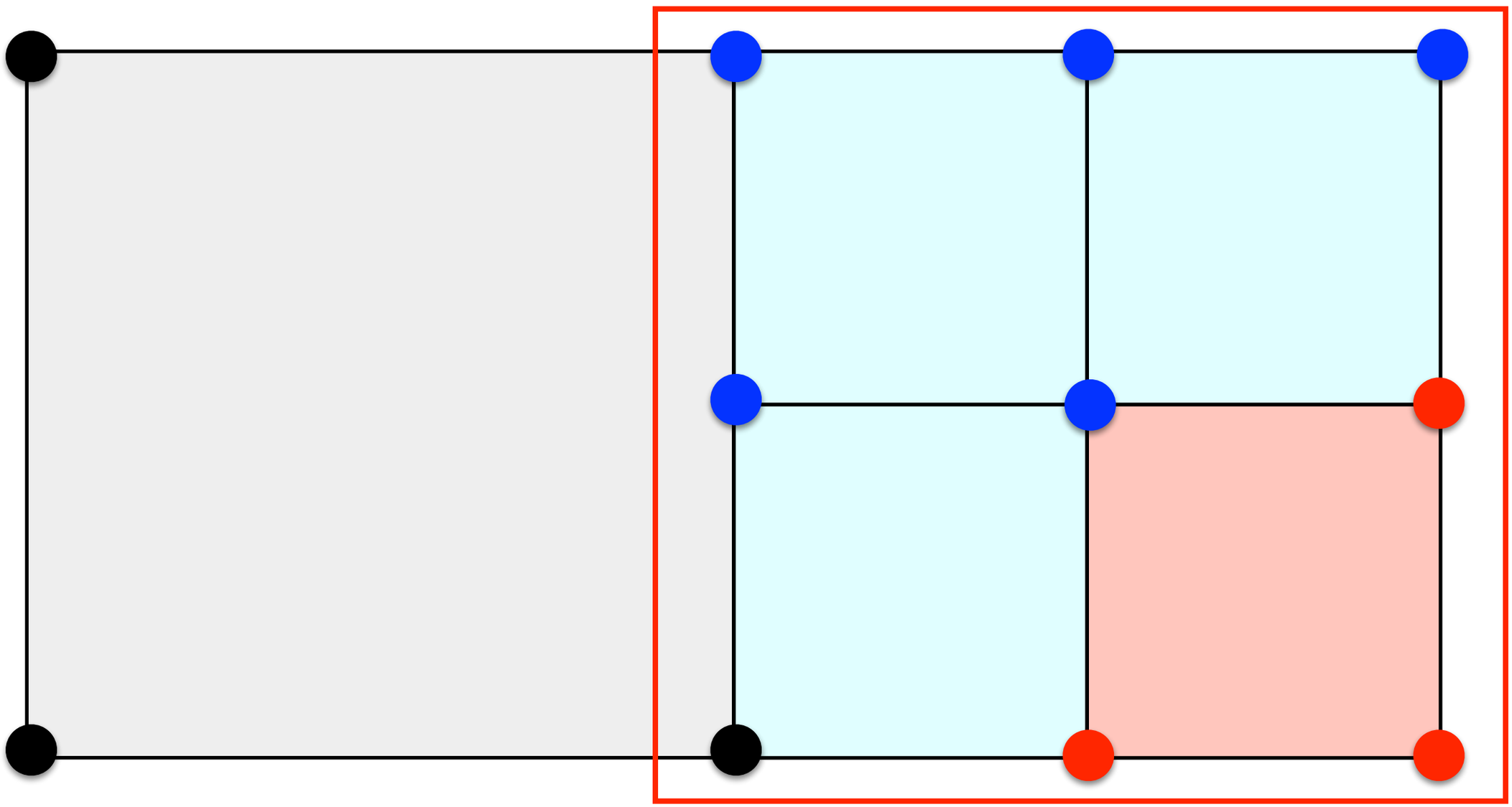
- Required for Kelly error estimator

Distribution of DoFs:
Locally relevant degrees-of-freedom (process 1)



- Required for Kelly error estimator

Distribution of DoFs:
Locally relevant degrees-of-freedom (process 2)



- Required for Kelly error estimator

Changes: New headers

- MPI
  `#include <deal.II/base/mpi.h>`
- Parallel shared triangulation
  `#include <deal.II/distributed/shared_tria.h>`
- Filtered iterator
  `#include <deal.II/grid/filtered_iterator.h>`
- IndexSet
  `#include <deal.II/base/index_set.h>`
- Trilinos linear algebra
  `#include <deal.II/lac/trilinos_*>`
- Output filter
  `#include <deal.II/base/conditional_ostream.h>`

Changes: Class definition

- MPI utility objects
  - MPI communicator
  - Number of processes, number of “this” process
  - Stream output assistant (filter)
- Triangulation type
- Sparse linear algebra objects
- IndexSets
  - Locally owned
  - Locally relevant

Changes: System setup

- Must determine the set of locally owned and locally relevant DoFs
  - Locally owned = those assigned to a particular MPI process
  - Locally relevant = those assigned to other processors, but are required to perform some action on the current process
- Distribution of sparsity pattern
  - Tell the locally defined sparsity pattern which entries require data exchange / may be written into by other processes
- Can interrogate information about problem distribution as viewed from other processors

Changes: Assembly

- Cell loop: Only cells owned by the MPI process
  - Can use filtered iterators
  - Can check within the cell loop
    `cell->locally_owned();`
- All assembled data is initially localised to a process
- Final synchronisation of data between MPI processes
  - Accumulation of values on DoFs written into by more than one process
    `[matrix/vector].compress(VectorOperations::add);`
    - Only once (outside of cell loop)

Changes: Linear solver

- Solver templated on Vector type
- Preconditioner type

Changes: Mesh refinement

- Kelly error estimator needs to know solution values on cells that are not owned by this current MPI process

- Need to provide view of solution with values for all locally owned and locally relevant DoFs

Changes: Postprocessing

- Write out portion of solution from each processor
  - deal.II writes these outputs on a per-cell basis
  - Need to provide view of solution with values for all locally owned and locally relevant DoFs

Changes: Main function

- Setup MPI environment

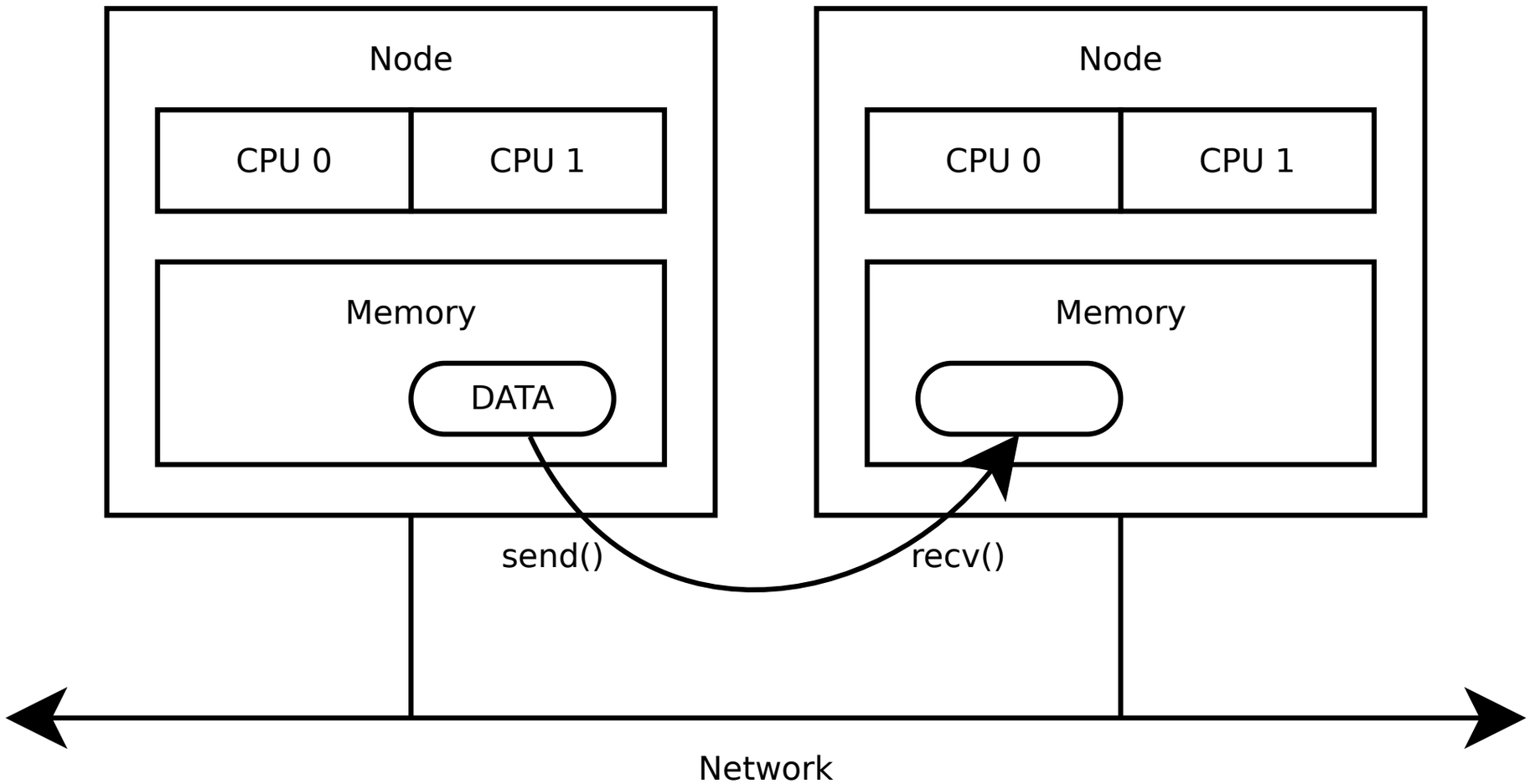
```
Utilities::MPI::MPI_InitFinalize
mpi_initialization(argc, argv, 1);
```

MPI Parallelization, Part II
parallel::distributed::Triangulation

Jean-Paul Pelteret (jean-paul.pelteret@fau.de)
Luca Heltai (luca.heltai@sissa.it)

21 March 2018

Parallel computing model: MPI



General Considerations

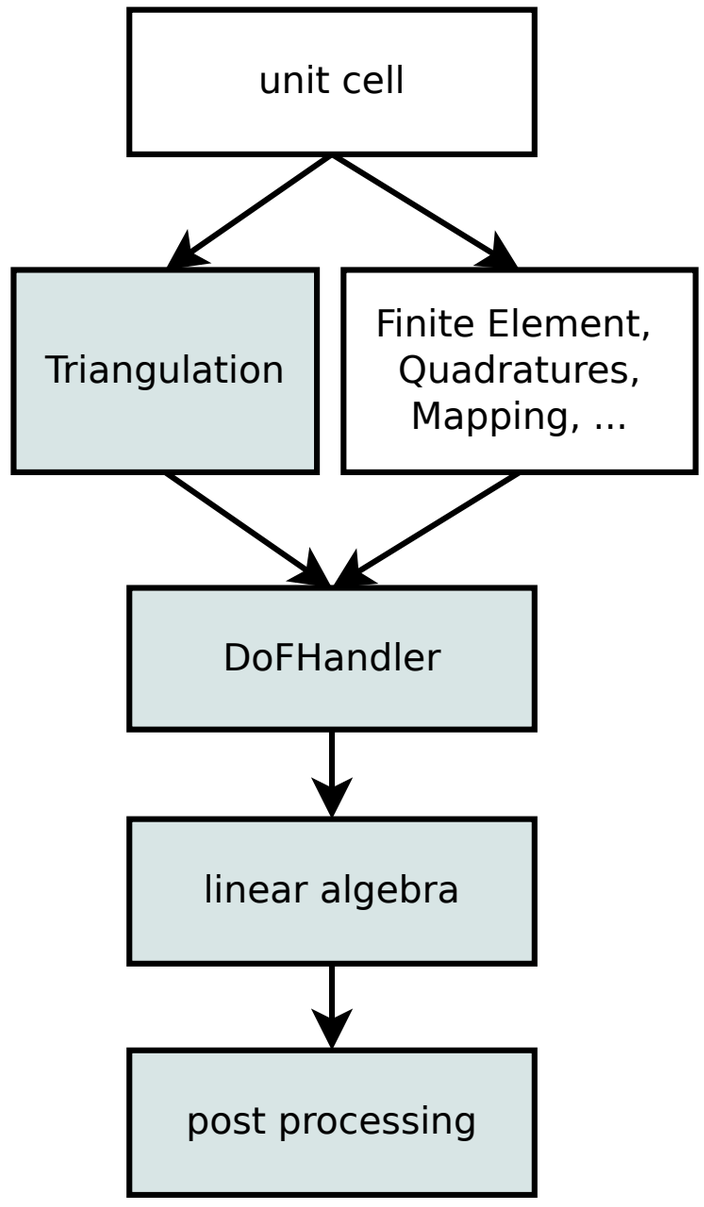
- Goal: get the solution faster!
- If FEM with <500.000 dofs, and 2d, use direct solver!

- If you need more, then you have to **SPLIT** the work
  - **Distributed data** storage everywhere
    - need special data structures
- **Efficient algorithms**
  - not depending on total problem size
- **“Localize” and “hide” communication**
  - point-to-point communication, nonblocking sends and receives

Master in High Performance Computing

Needs to be parallelized:

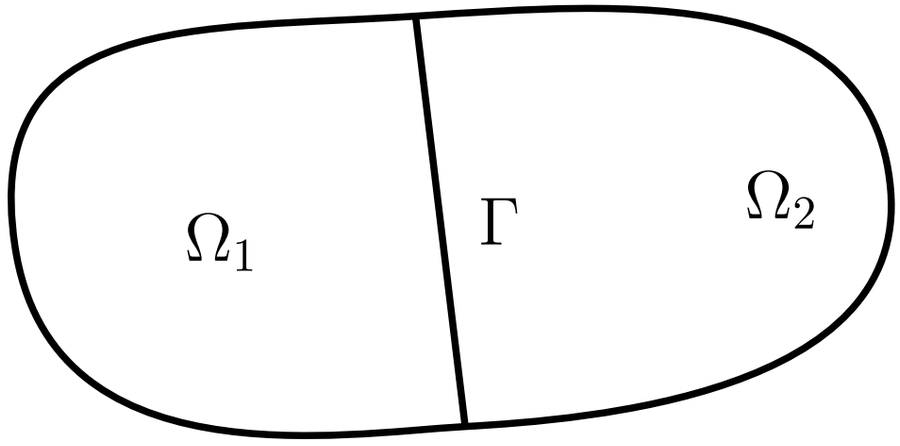
1. Triangulation (mesh with associated data)
   — hard: distributed storage, new algorithms
2. DoFHandler (manages degrees of freedom)
   — hard: find global numbering of DoFs
3. Linear Algebra (matrices, vectors, solvers)
   — use existing library
4. Postprocessing (error estimation, solution transfer, output, . . .)
   — do work on local mesh, communicate



SISSA ICTP

How to Parallelize?

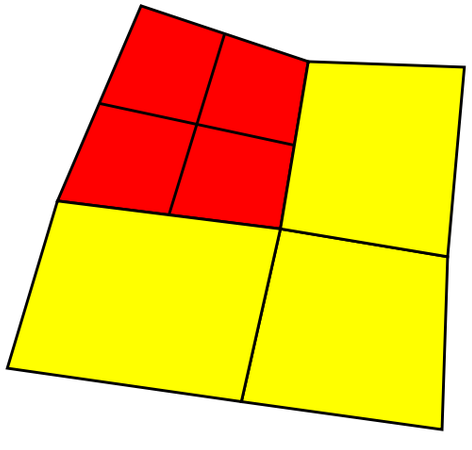
Option 1: Domain Decomposition



- Split up problem on PDE level
- Solve subproblems independently
- Converges against global solution
- Problems:
  - Boundary conditions are problem dependent:
    $\rightsquigarrow$ sometimes difficult!
    $\rightsquigarrow$ no black box approach!
  - Without coarse grid solver:
    condition number grows with # subdomains
    $\rightsquigarrow$ no linear scaling with number of CPUs!

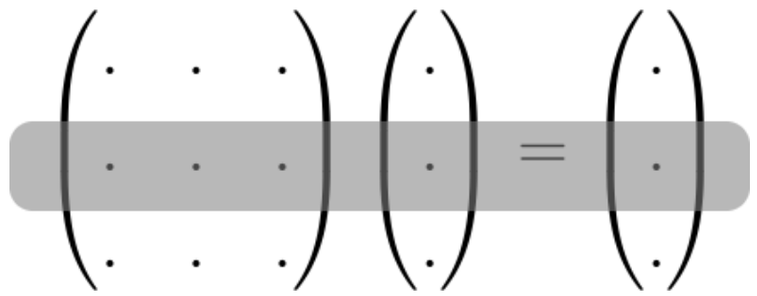
How to Parallelize?

Option 2: Algebraic Splitting



- Split up mesh between processors:

- Assemble logically global linear system (distributed storage):

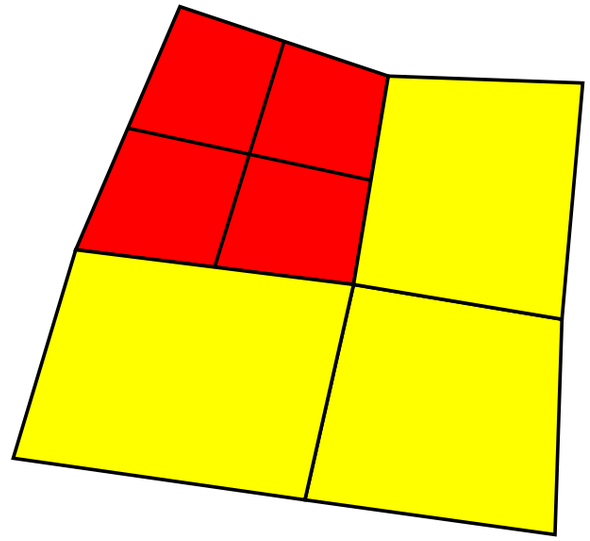


- Solve using iterative linear solvers in parallel
- Advantages:
  - Looks like serial program to the user
  - Linear scaling possible (with good preconditioner)

Partitioning

Optimal partitioning (coloring of cells):

- same size per region
  $\rightsquigarrow$ even distribution of work
- minimize interface between region
  $\rightsquigarrow$ reduce communication



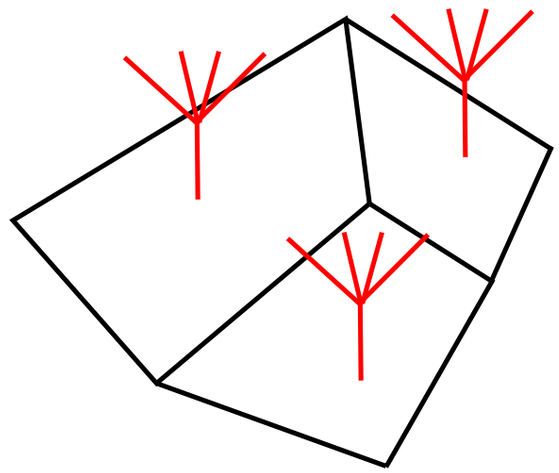
Optimal partitioning is an NP-hard graph partitioning problem.

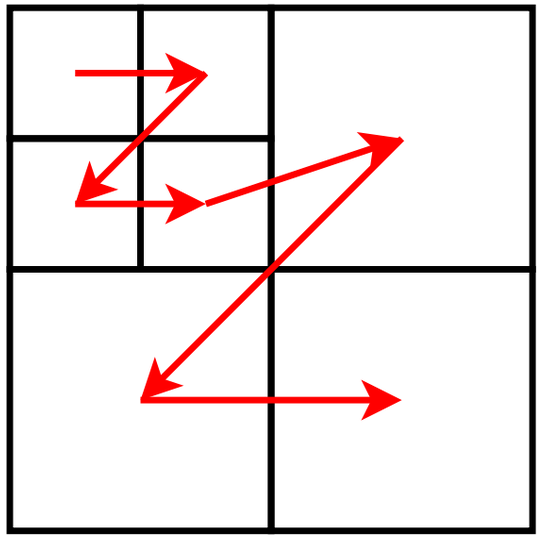
- Typically done: heuristics (existing tools: METIS)
- Problem: worse than linear runtime
- Large graphs: several minutes, memory restrictions

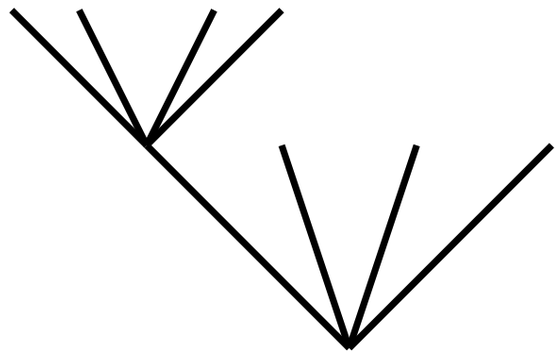
$\rightsquigarrow$ Alternative: avoid graph partitioning

Partitioning with “Space filling curves”

- *p4est* library: parallel quad-/octrees
- Store refinement flags from a base mesh
- Based on space-filling curves
- Very good scalability



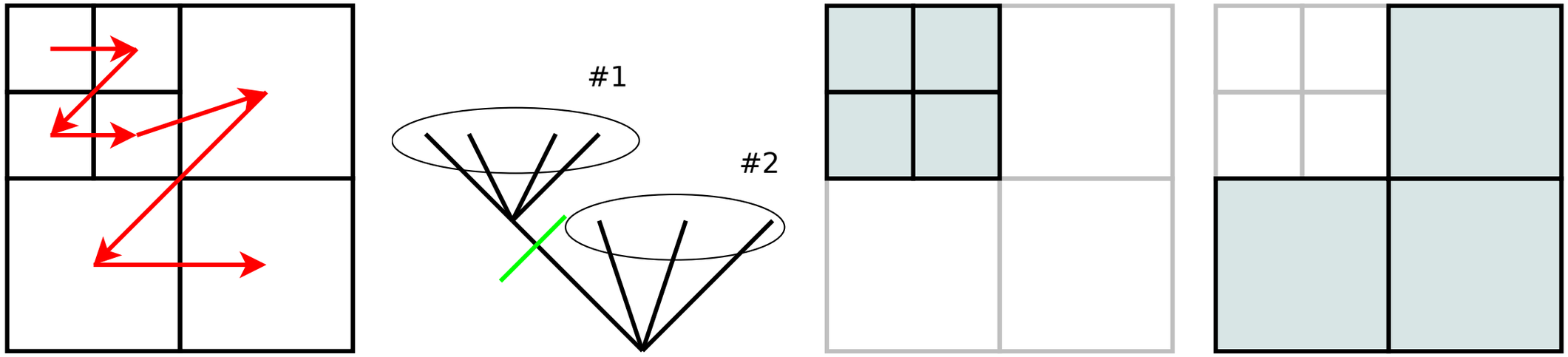




Burstedde, Wilcox, and Ghattas.
p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees.
SIAM J. Sci. Comput., 33 no. 3 (2011), pages 1103-1133.

Triangulation

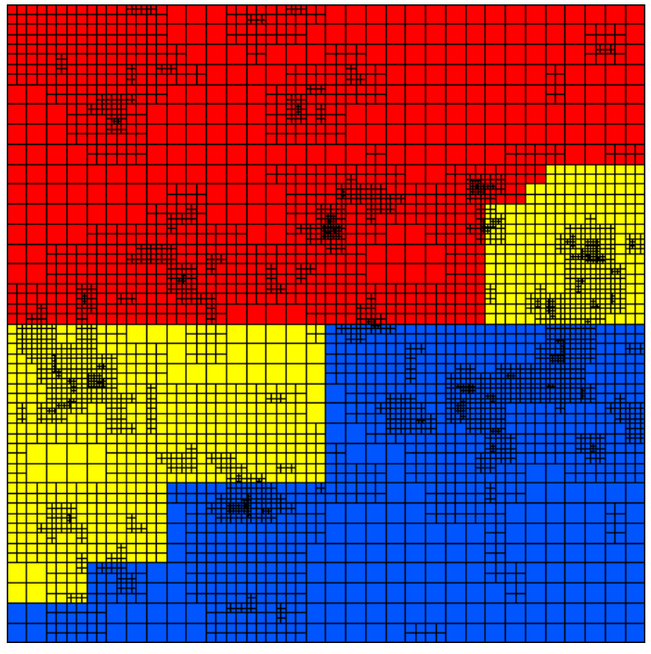
- Partitioning is cheap and simple:



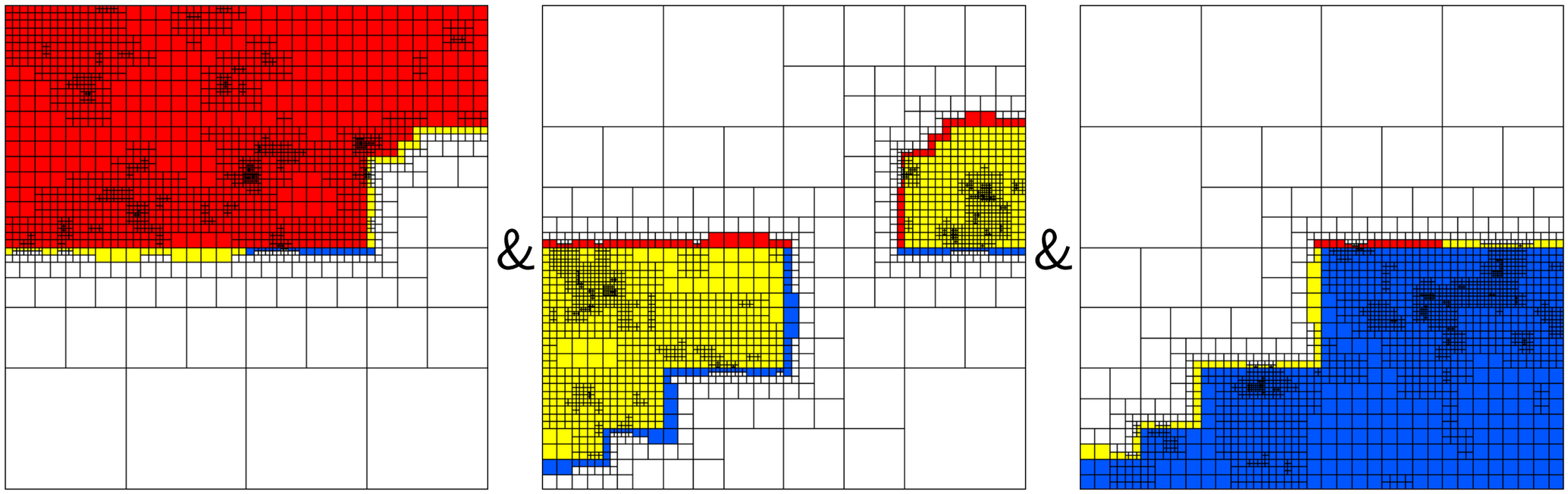
- Then: take *p4est* refinement information
- Recreate rich *deal.II* Triangulation only for local cells (stores coordinates, connectivity, faces, materials, . . .)
- How? recursive queries to *p4est*
- Also create ghost layer (one layer of cells around own ones)

MHPC
Master in High Performance Computing

Example (color by CPU ID)



=



SISSA

ICTP

What's needed?

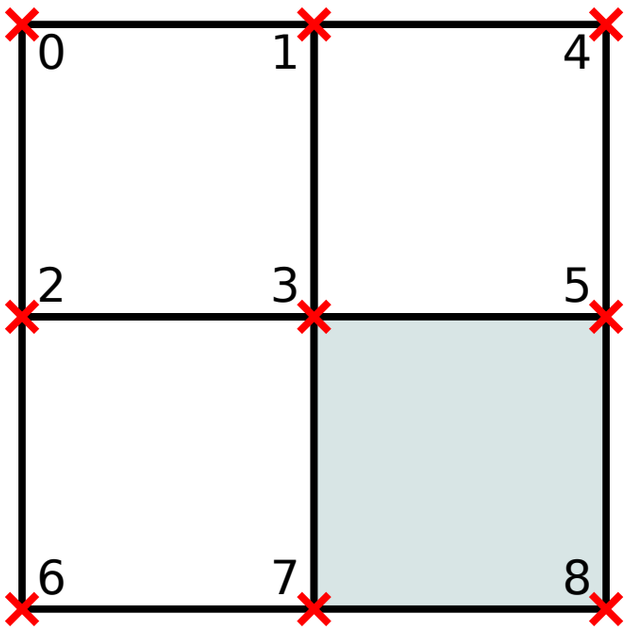
How to use?

- Replace `Triangulation` by `parallel::distributed::Triangulation`
- Continue to load or create meshes as usual
- Adapt with `GridRefinement::refine_and_coarsen*` and `tr.execute_coarsening_and_refinement()`, etc.
- You can only look at own cells and ghost cells: `cell->is_locally_owned()`, `cell->is_ghost()`, or `cell->is_artificial()`
- Of course: dealing with DoFs and linear algebra changes!

What's needed?

	serial mesh	dynamic parallel mesh	static parallel mesh
name	Triangulation	parallel::distributed ::Triangulation	(just an idea)
duplicated	everything	coarse mesh	nothing
partitioning	METIS	p4est: fast, scalable	offline, (PAR)METIS?
part. quality	good	okay	better?
hp?	yes	(planned)	yes?
geom. MG?	yes	in progress	?
Aniso. ref.?	yes	no	(offline only)
Periodicity	yes	in progress	?
Scalability	100 cores	16k+ cores	?

Sketch...

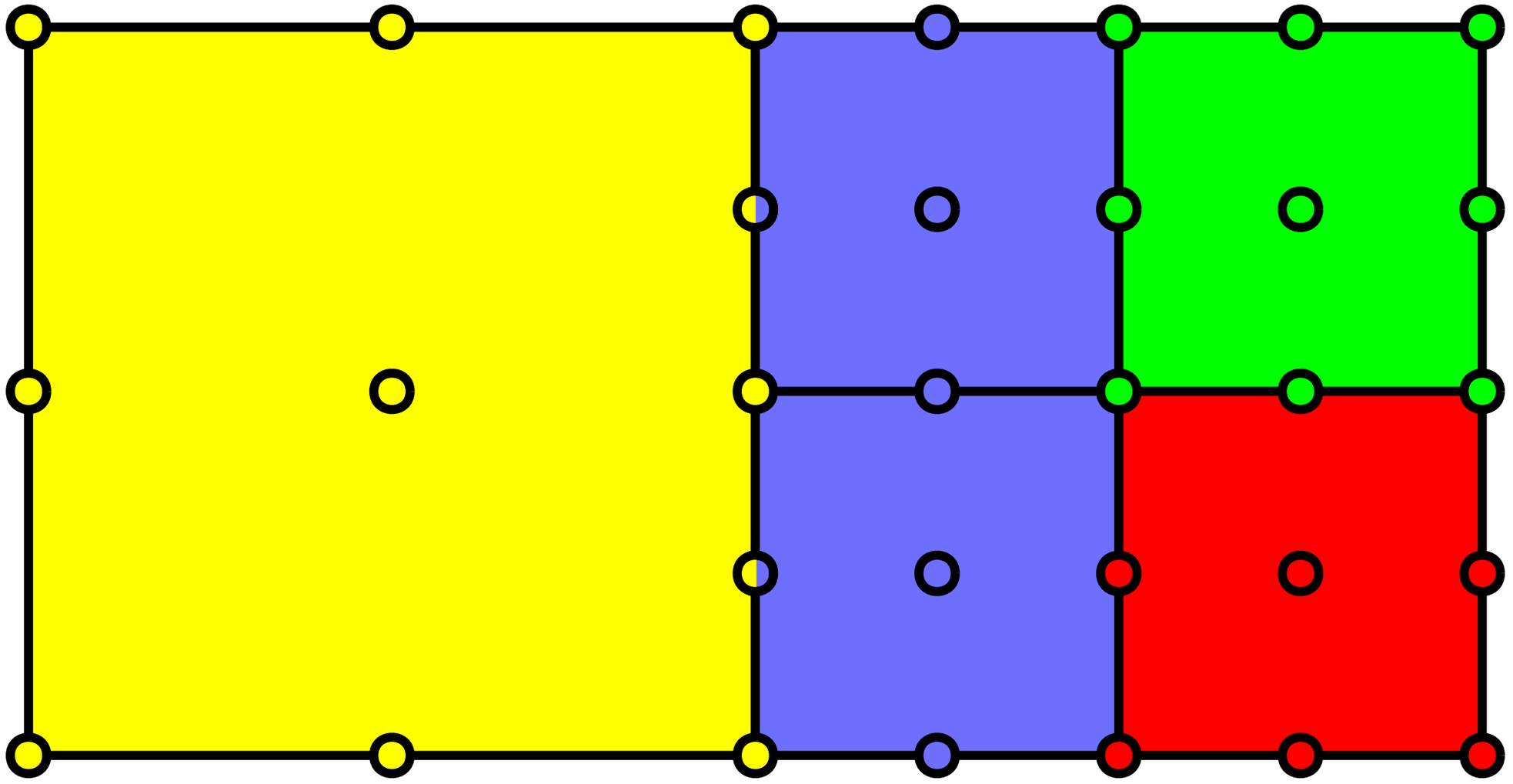


- Create global numbering for all DoFs
- Reason: identify shared ones
- Problem: no knowledge about the whole mesh

Sketch:

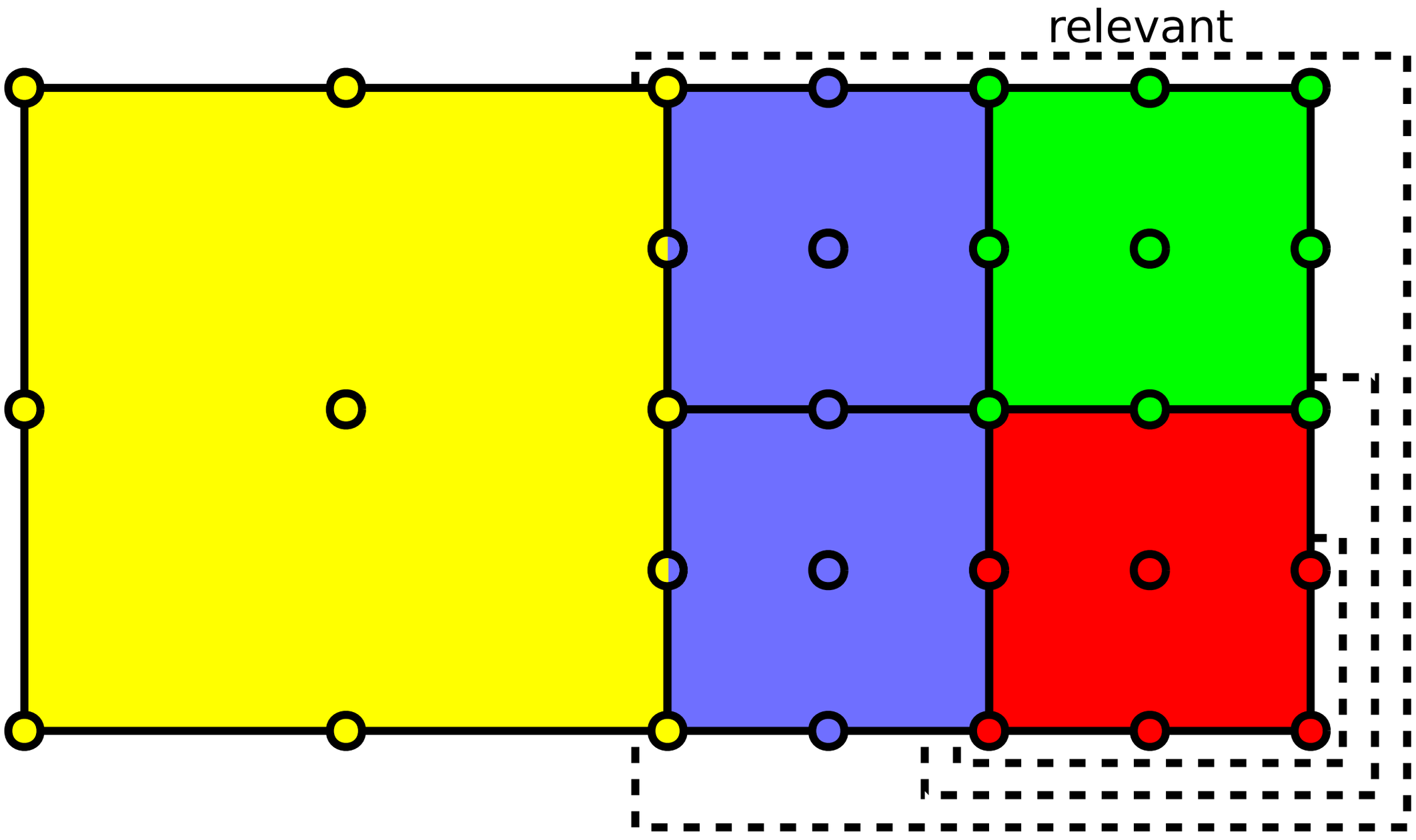
1. Decide on ownership of DoFs on interface (no communication!)
2. Enumerate locally (only own DoFs)
3. Shift indices to make them globally unique (only communicate local quantities)
4. Exchange indices to ghost neighbors

Sketch...



- Example: Q2 element and ownership of DoFs
- What might red CPU be interested in?

Sketch...



(perspective of the red CPU)

Sketch...

- Each CPU has sets:
  - `owned`: we store vector and matrix entries of these rows
  - `active`: we need those for assembling, computing integrals, output, etc.
  - `relevant`: error estimation
- These set are subsets of $\{0, \ldots,$`n_global_dofs`$\}$
- Represented by objects of type `IndexSet`
- How to get? `DoFHandler::locally_owned_dofs()`, `DoFTools::extract_locally_relevant_dofs()`, `DoFHandler::locally_owned_dofs_per_processor()`, ...

Sketch...

- reading from owned rows only (for both vectors and matrices)
- writing allowed everywhere (more about `compress` later)
- what if you need to read others?
- Never copy a whole vector to each machine!
- instead: ghosted vectors

Sketch...

- read-only
- create using
  `Vector(IndexSet owned, IndexSet ghost, MPI_COMM)`
  where ghost is `relevant` or `active`
- copy values into it by using operator=(Vector)
- then just read entries you need

Compressing

- Why?
  - After writing into foreign entries communication has to happen
  - All in one go for performance reasons
- How?
  - `object.compress (VectorOperation::add);` if you added to entries
  - `object.compress (VectorOperation::insert);` if you set entries
  - This is a collective call
- When?
  - After the assembly loop (with `::add`)
  - After you do `vec(j) = k;` or `vec(j) += k;` (and in between add/insert groups)
  - In no other case (all functions inside deal.II compress if necessary)! (this is new!)

Trilinos VS PETSc

What should I use?

- Similar features and performance
- Pro Trilinos: more development, some more features (automatic differentation, . . .), cooperation with deal.II
- Pro PETSc: stable, easier to compile on older clusters
- But: being flexible would be better! – “why not both?”
  - you can! Example: new `step-40`
  - can switch at compile time
  - need #ifdef in a few places (different solver parameters TrilinosML vs BoomerAMG)
  - some limitations, somewhat work in progress

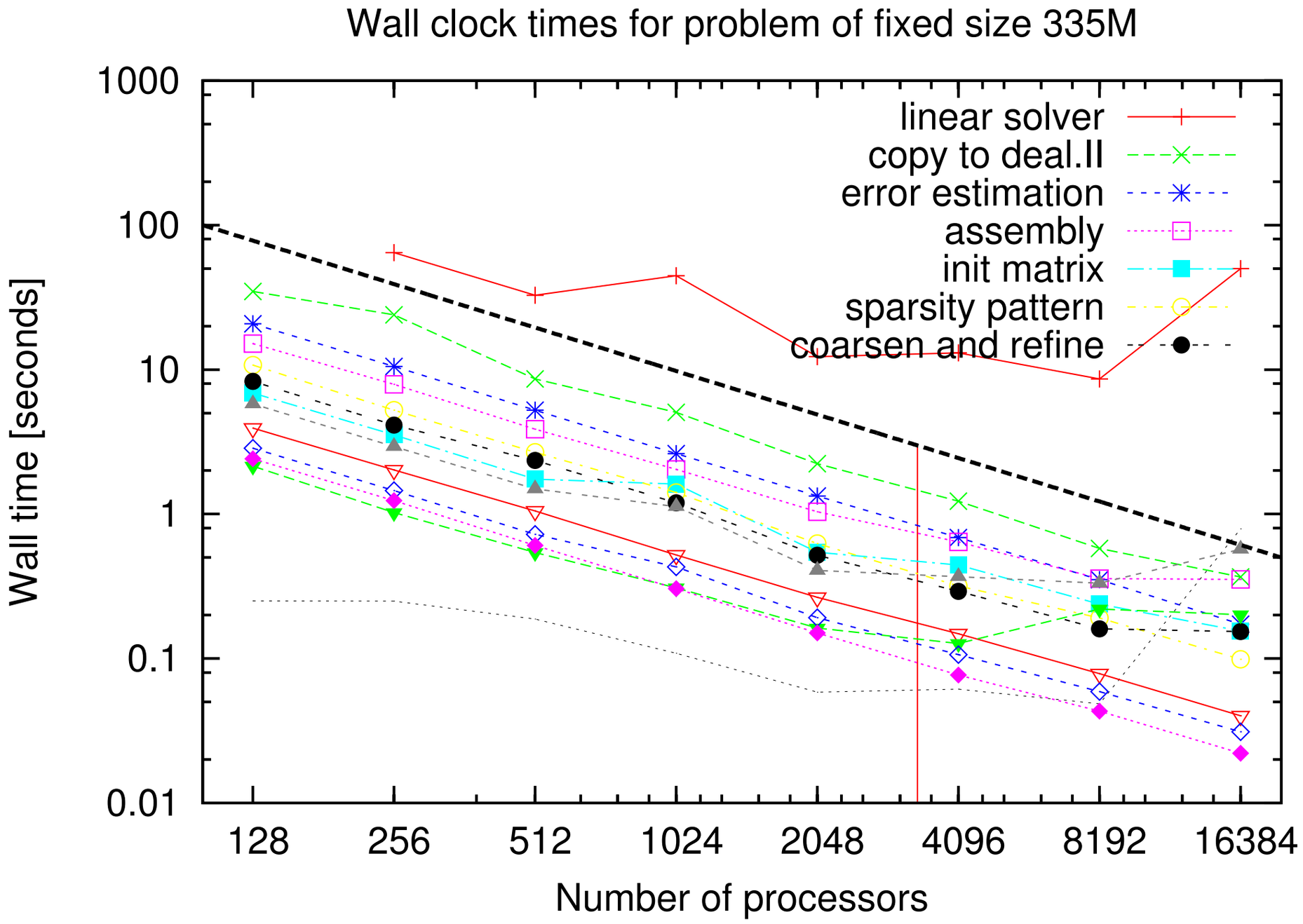
Trilinos VS PETSc

```
#include <deal.II/lac/generic_linear_algebra.h>
#define USE_PETSC_LA // uncomment this to run with Trilinos

namespace LA
{
#ifdef USE_PETSC_LA
  using namespace dealii::LinearAlgebraPETSc;
#else
  using namespace dealii::LinearAlgebraTrilinos;
#endif
}

// ...
    LA::MPI::SparseMatrix system_matrix;
    LA::MPI::Vector solution;

// ...
    LA::SolverCG solver(solver_control, mpi_communicator);
    LA::MPI::PreconditionAMG preconditioner;

    LA::MPI::PreconditionAMG::AdditionalData data;

#ifdef USE_PETSC_LA
    data.symmetric_operator = true;
#else
    //trilinos defaults are good
#endif
    preconditioner.initialize(system_matrix, data);

// ...
```

Solvers

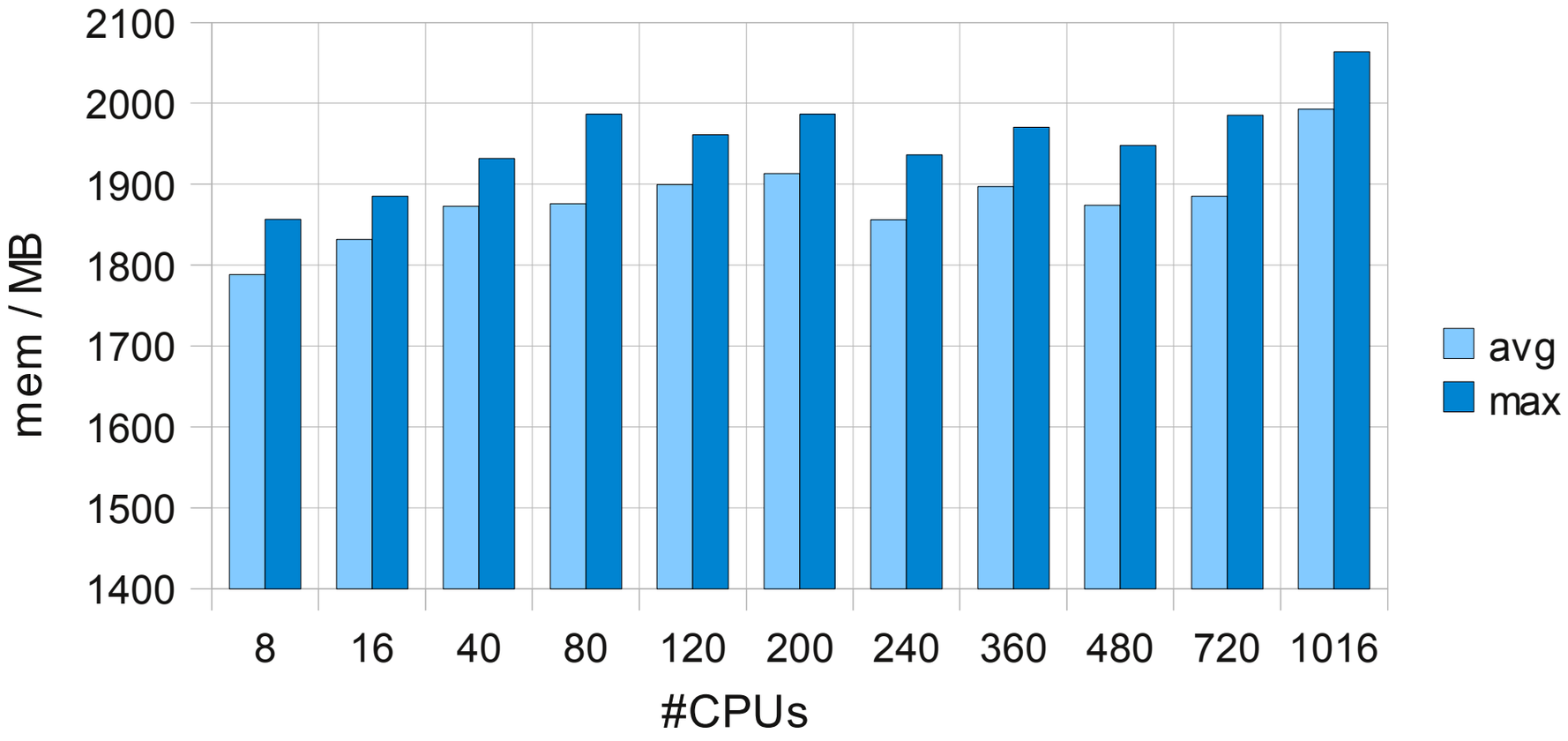
- Iterative solvers only need Mat-Vec products and scalar products
  $\rightsquigarrow$ equivalent to serial code
- Can use templated deal.II solvers like GMRES!
- Better: use tuned parallel iterative solvers that hide/minimize communication
- Preconditioners: more work, just operating on local blocks not enough

Strong Scaling: 2d Poisson



Master in High Performance Computing

Memory Consumption



average and maximum memory consumption (VmPeak)
3D, weak scalability from 8 to 1000 processors with about 500.000 DoFs per processor (4 million up to 500 million total)

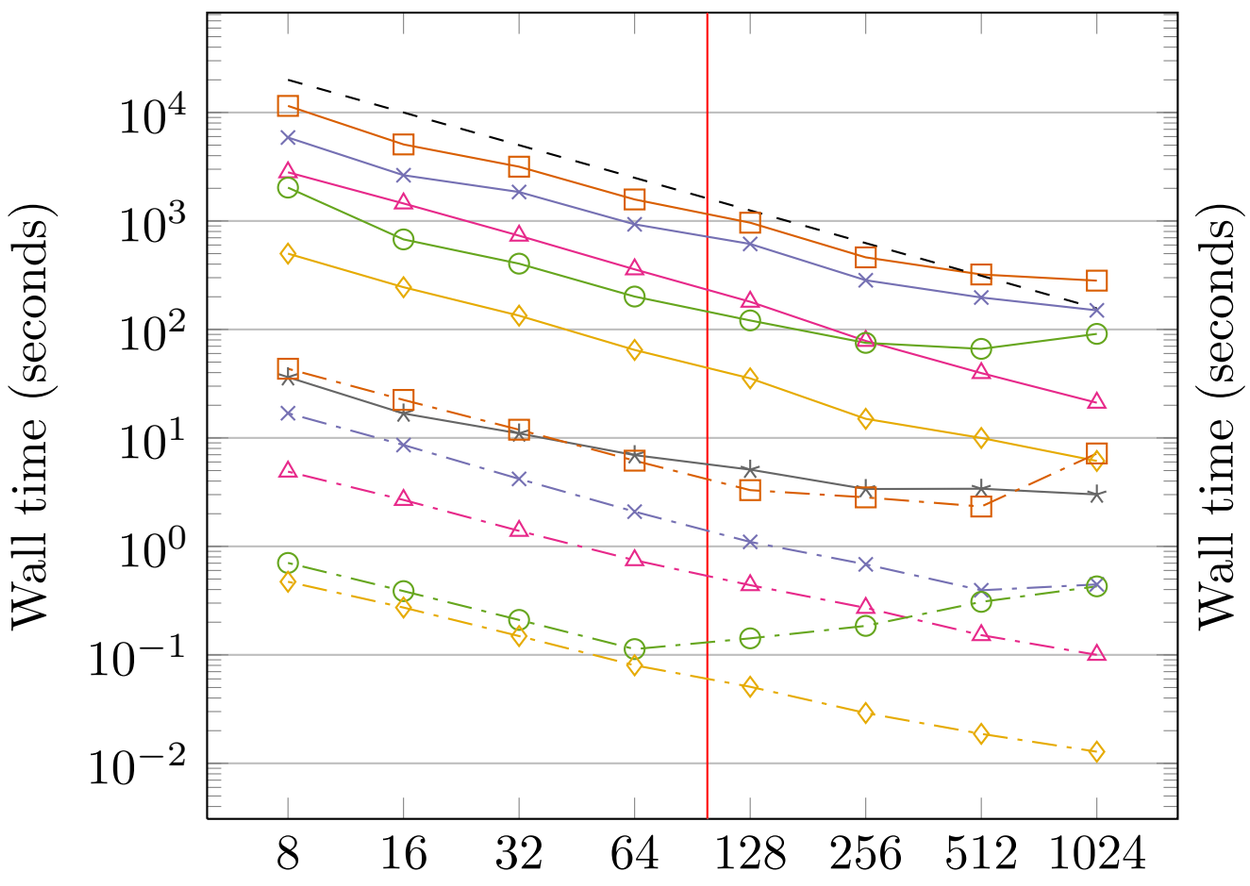
$\rightsquigarrow$ **Constant memory usage with increasing # CPUs & problem size**

SISSA ICTP

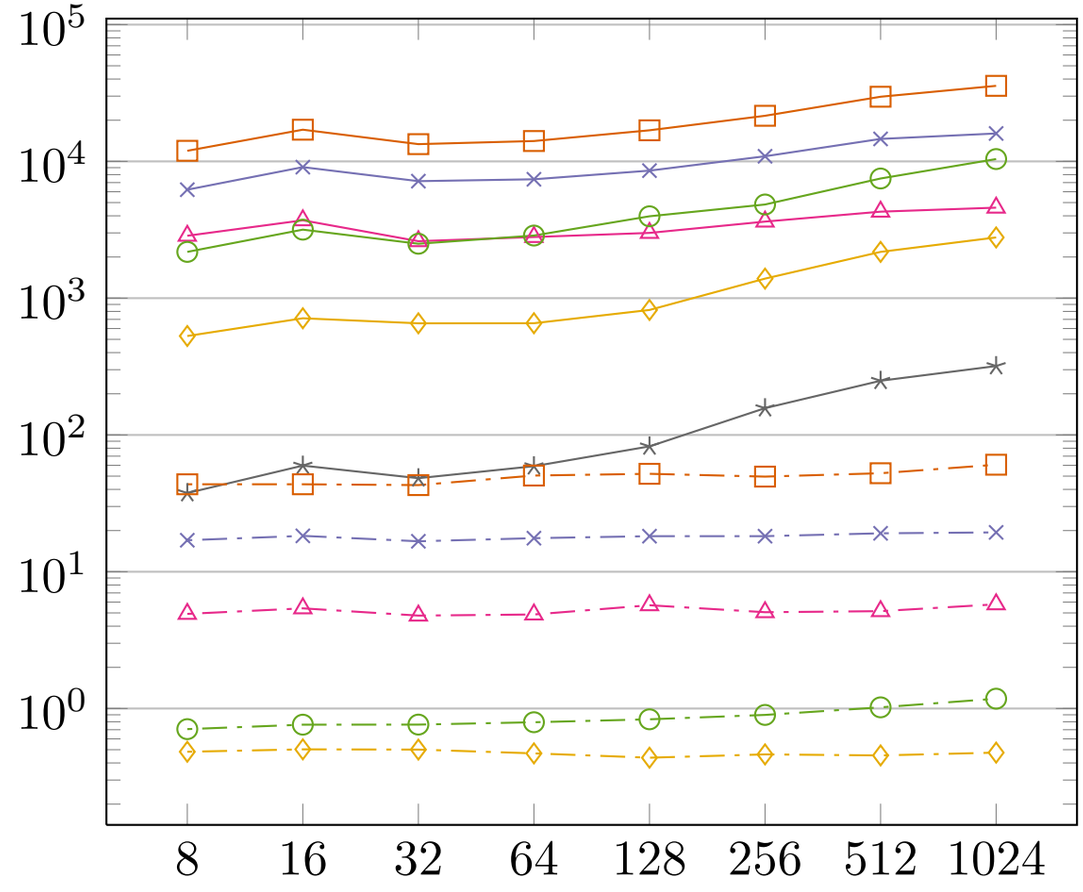
Step 40

Master in High Performance Computing

Strong Scaling (9.9M DoFs)



Weak Scaling (1.2M DoFs/Core)



TOTAL
Solve: iterate
Assembling
Solve: setup
Residual
update active set
Setup: refine mesh
Setup: matrix
Setup: distribute DoFs
Setup: vectors
Setup: constraints

SISSA

ICTP