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 schedule

Time	Duration	Content	Speaker	Content	Speaker
		MONDAY 25.02.2019		**TUESDAY 26.02.2019**	
09:30	1.25 hours	Introduction First steps	DD	Local refinement Hanging nodes - Part 1	DD
		COFFEE / TEA			
11:15	1.25 hours	Introduction to FEM	LH	Exercises, Q&A	DD, LH
		LUNCH			
14:00	1.25 hours	Solving Poisson's equation	DD	Local refinement Hanging nodes - Part 2	LH
		COFFEE / TEA			
15:45	1.25 hours	Exercises, Q&A	DD, LH	Exercises, Q&A	DD, LH
		WEDNESDAY 27.02.2019		**THURSDAY 28.02.2019**	
09:30	1.25 hours	Shared memory parallelisation	DD	MPI parallelisation: Part 1	DD
		COFFEE / TEA			
11:15	1.25 hours	Exercises, Q&A	DD, LH	Exercises, Q&A	DD, LH
		LUNCH			
14:00	1 hour	Denis' Talk (SISSA Main A-133)	DD	MPI parallelisation: Part 2	LH
		COFFEE / TEA			
15:45	1.25 hours			Exercises, Q&A	DD, LH
		FRIDAY 01.03.2019		Project	

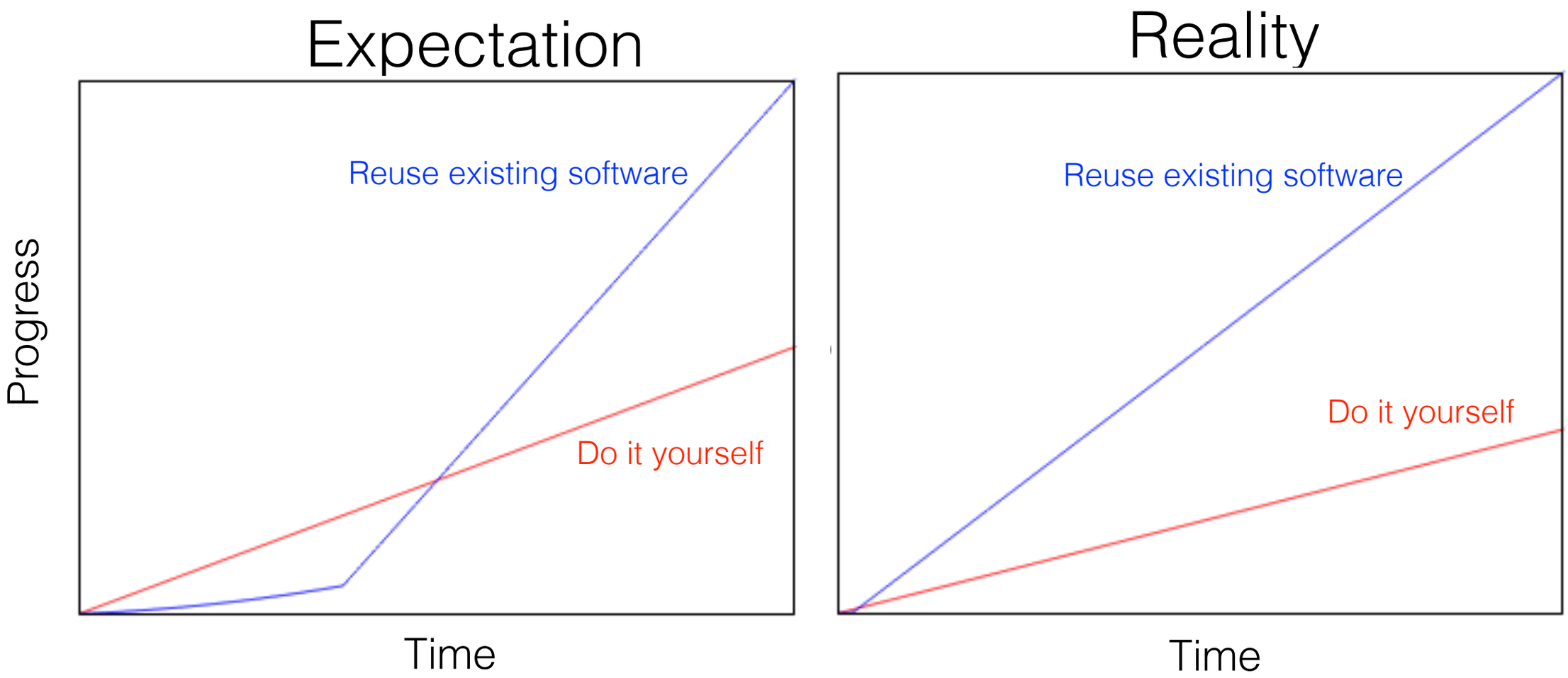
How the course will be run

- 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

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

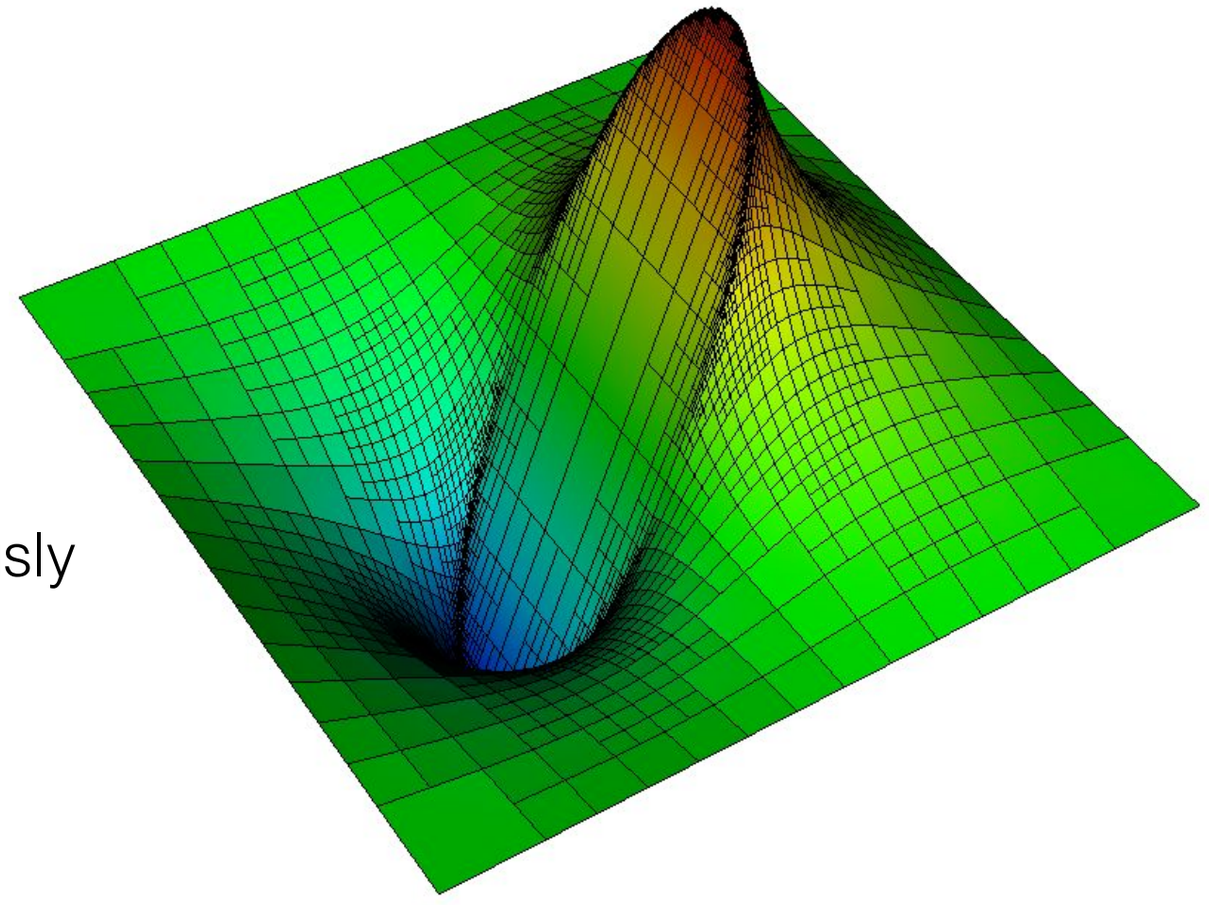


- Applies to:
  - Users
  - Developers
- “The secret to good scientific software is (re)using existing libraries”

What is deal.II?

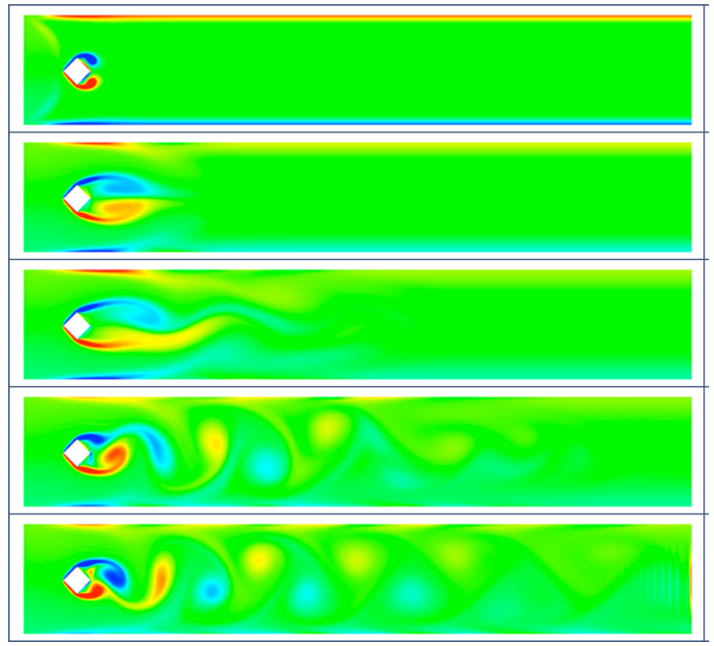
Differential Equation Analysis Library

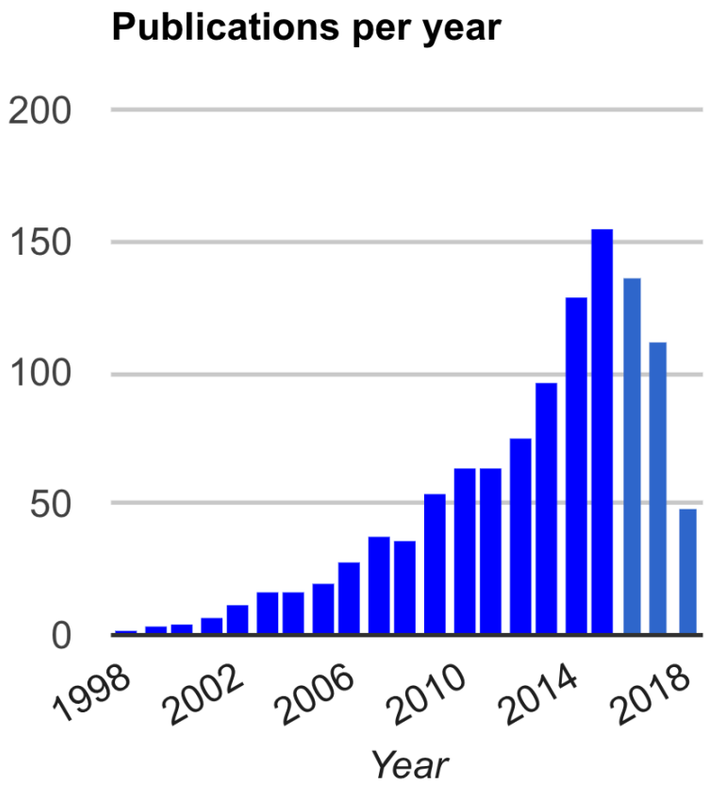
- Flexible open-source finite element toolkit
  - All the support functionality required to describe and solve a FE problem (PDEs)
  - Optimized for speed
  - Heavily tested
    - Many error checks (debug mode)
    - +10,000 regression tests run continuously
  - Part of SPEC CPU 2017 benchmark
- Templated C++ library (Object Orientated)
  - Dimension independent programming
- Portable
  - OS, architecture, compiler



What is deal.II?

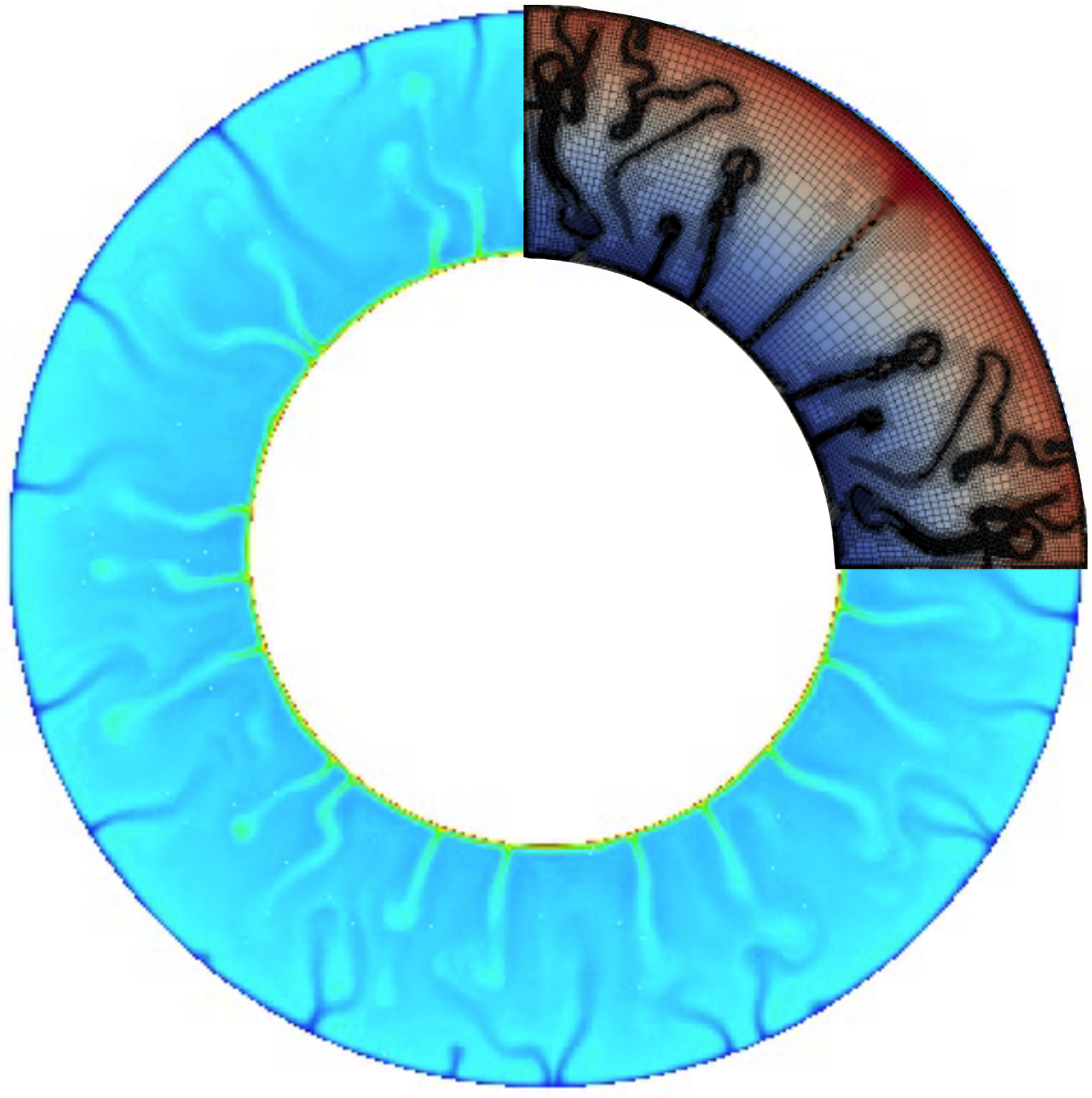
- Heavily documented
  - Over 10000 pages of interface documentation
  - Numerous tutorials
    - Illustrate functionality
    - Present methods to solve problems
- Quite widely used, and growing
- Active community
  - Approachable developers
  - Helpful online forum





Classes of problems solved using deal.II

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

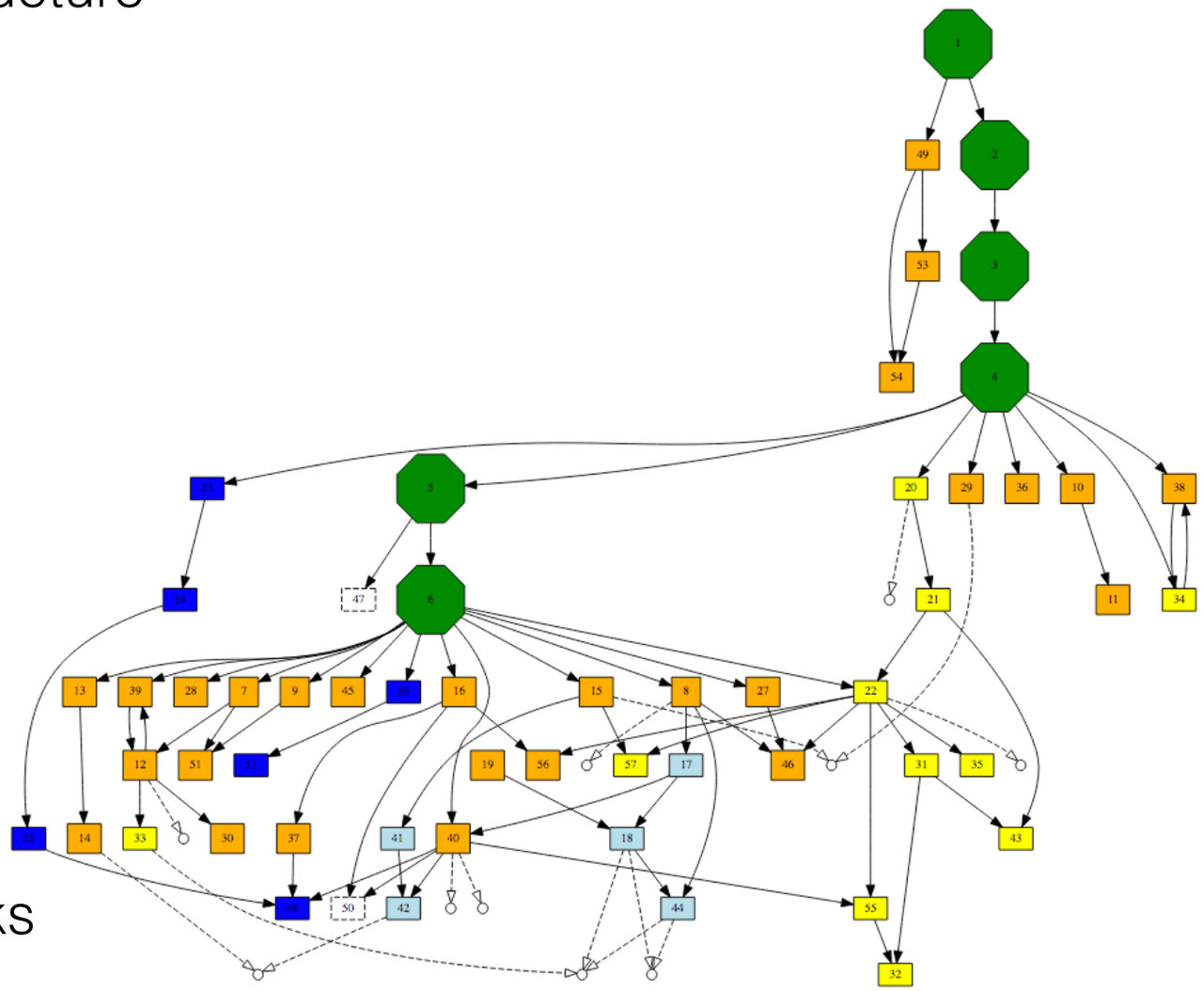


What deal.II is not

- 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 details, i.e.
    - Preconditioners
    - Constitutive equations

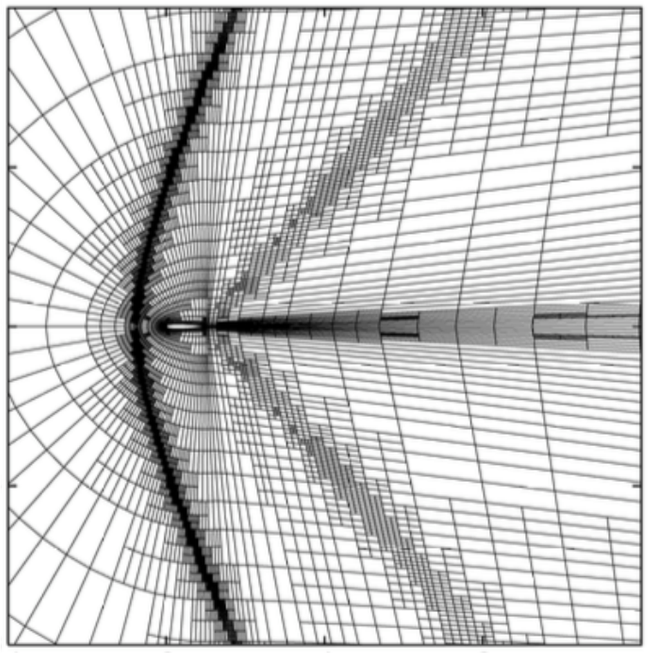
How deal.II will help you

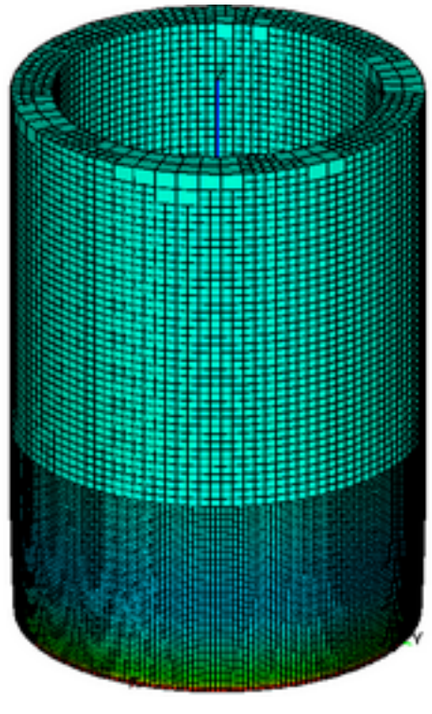
- Unified and well thought out data structure
  - Problem implementation
- Many tutorials
  - Baseline from which to build on
  - Demonstrate how to use features
- Comprehensive debugging support
  - Error messages everywhere!
- Some built in numerical tools
- Integration with advanced frameworks
  -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

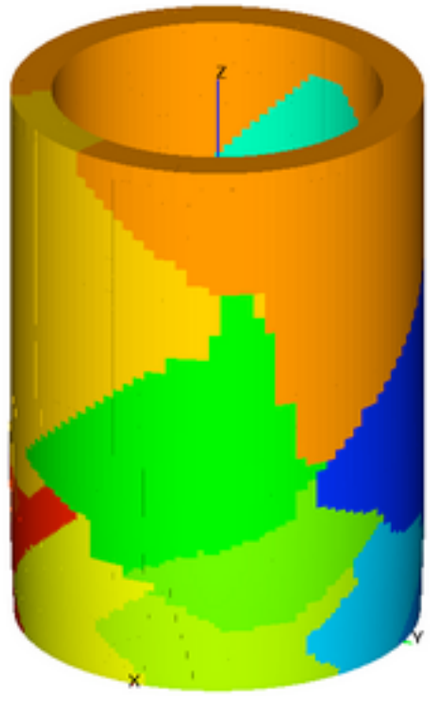


Fundamental capabilities and frameworks

- Mesh adaptivity
- Dense and sparse linear algebra
  - Built in tensor, dense matrix/vector classes
  - BLAS and LAPACK integration; GSL
  - Built in linear solvers and preconditioners
  - Eigenvalue solvers
- Parallelization
  - MPI
    - Linear algebra libraries (PETSc, Trilinos)
    - Distributed meshes ⟶ Billion DoFs
  - Threading (Intel TBB)
  - Vectorized numbers (AVX extensions)
- Pre/post-processing

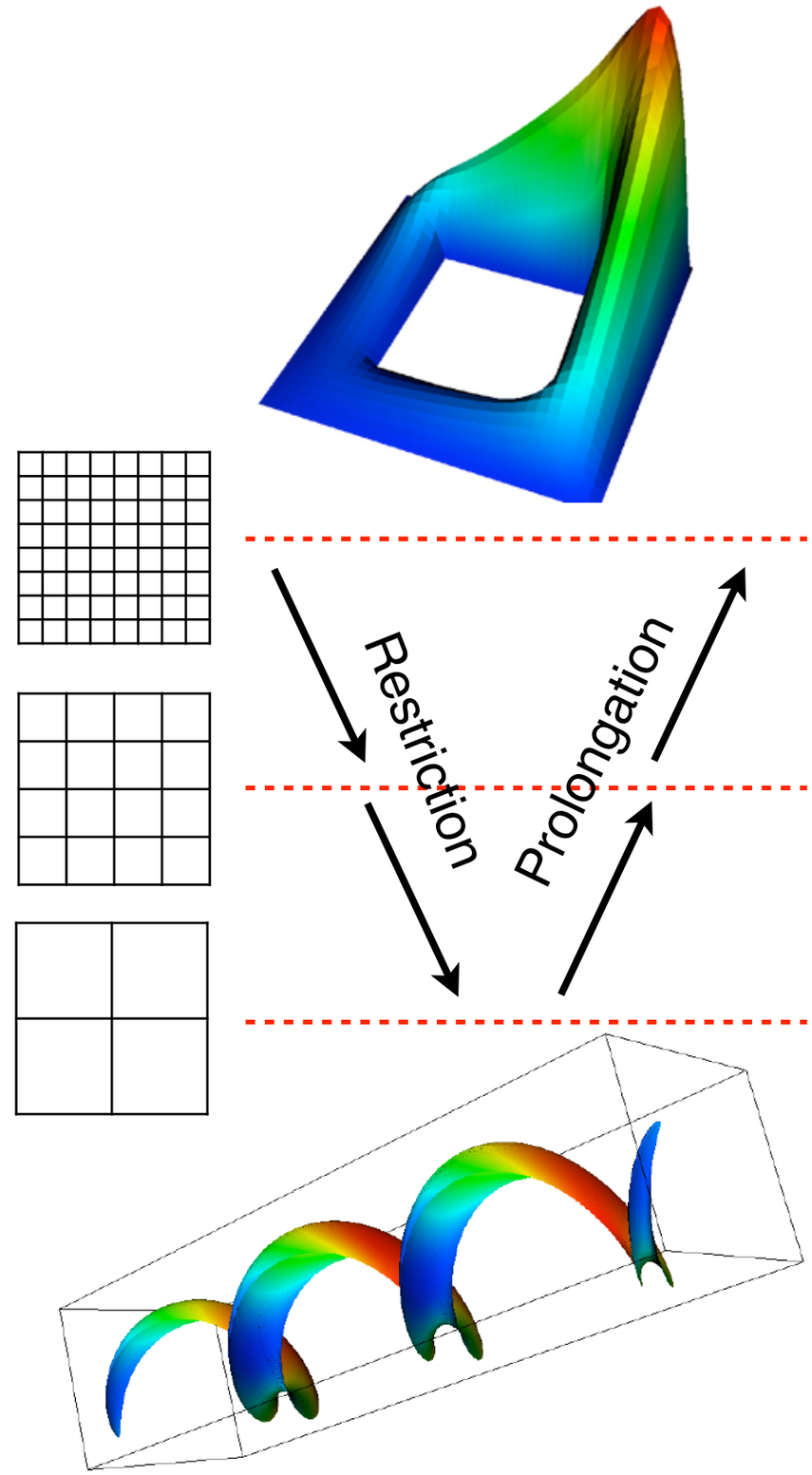






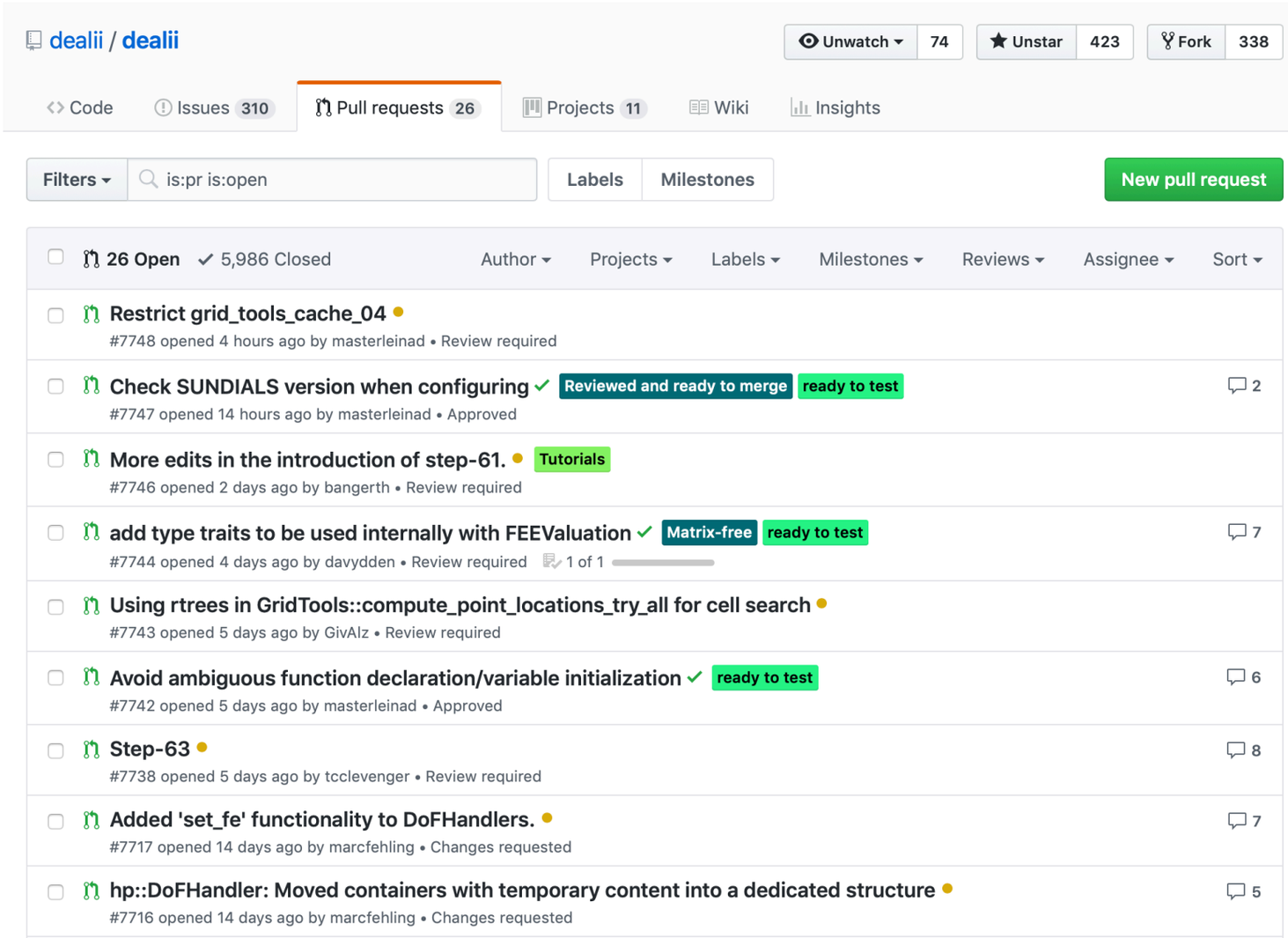
Advanced capabilities and frameworks

- 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
  - Exchange memory transfer for computations
- 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



Aims for this module

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

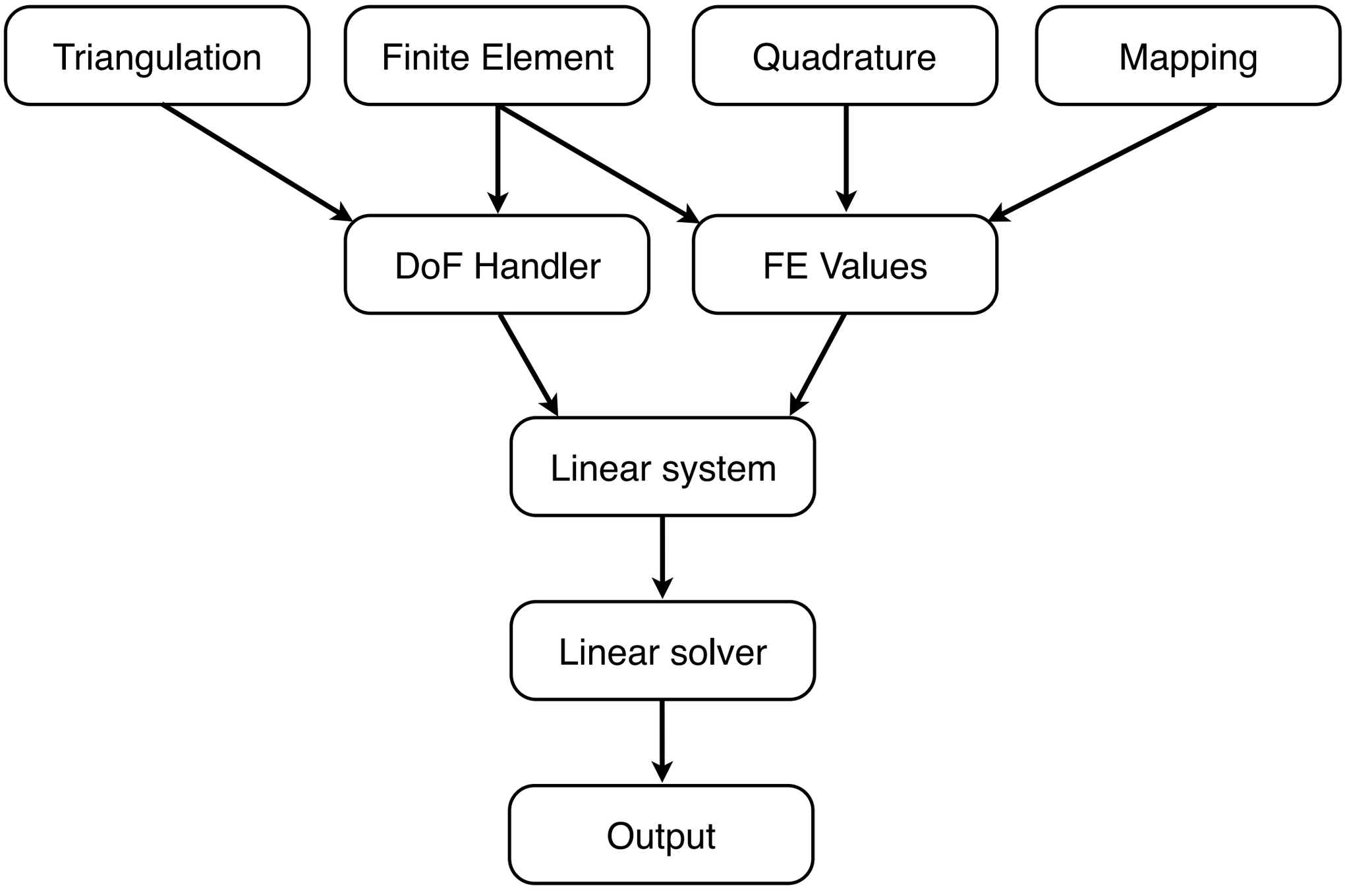
Reference material

- Main page
  https://dealii.org/9.0.0/doxygen/deal.II/index.html
- Tutorials
  - Step-1
    https://dealii.org/9.0.0/doxygen/deal.II/step_1.html
  - Step-49
    https://dealii.org/9.0.0/doxygen/deal.II/step_49.html
  - Step-2
    https://dealii.org/9.0.0/doxygen/deal.II/step_2.html

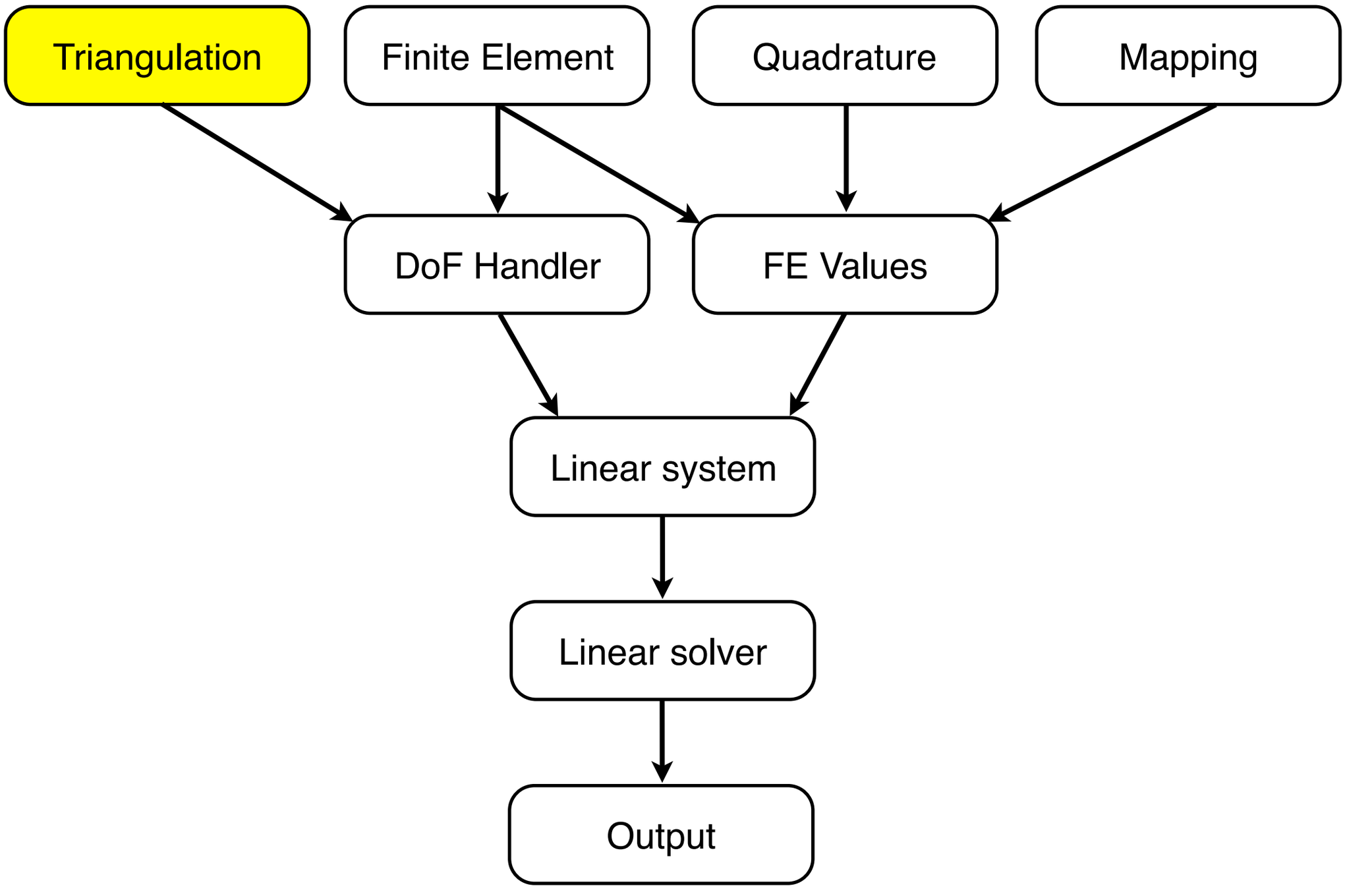
First and biggest tip

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

Structure of a prototypical FE problem

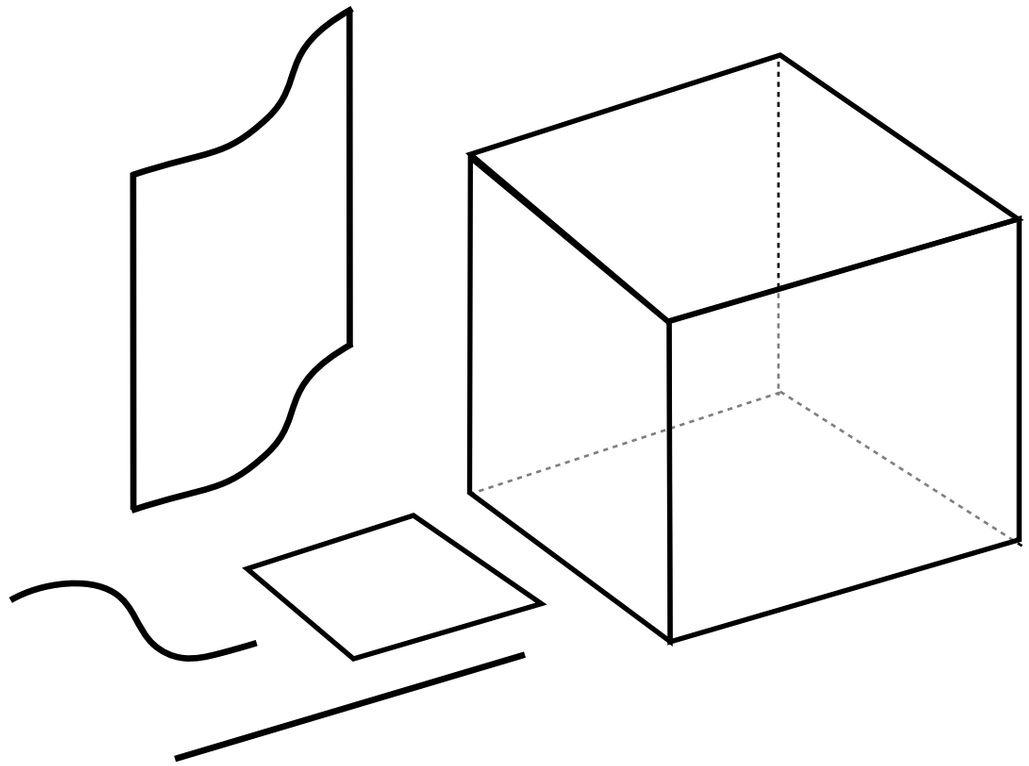


Structure of a prototypical FE problem



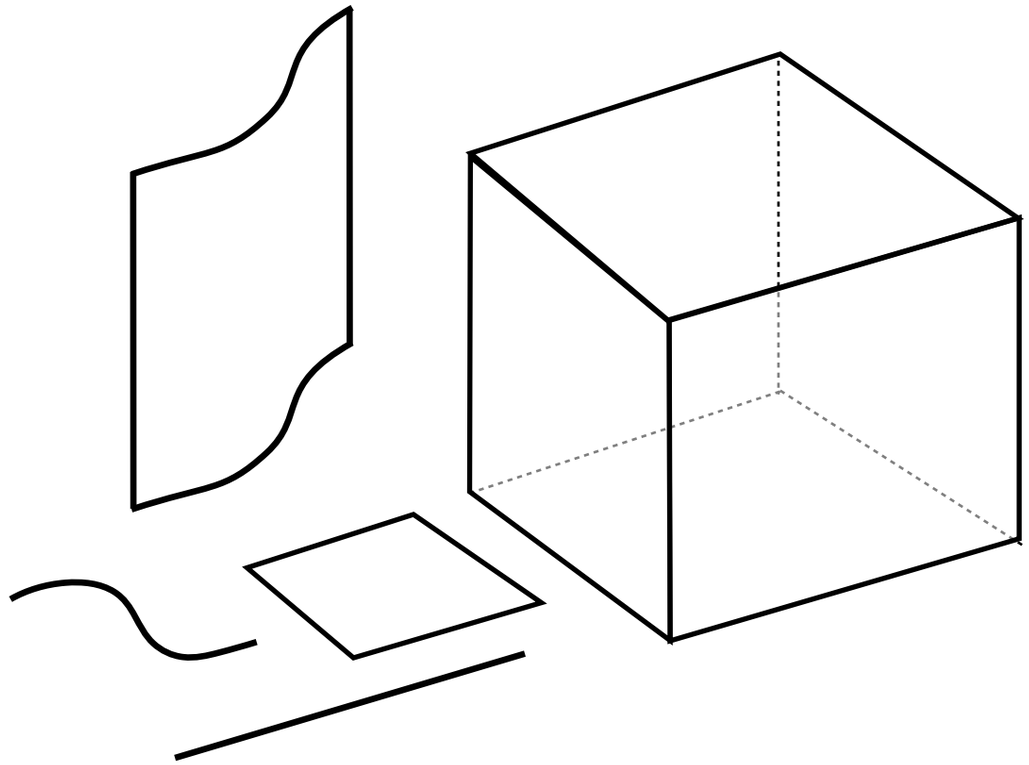
Interaction with geometry: the Triangulation class

- 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 (GridTools)
  - Can read in grids



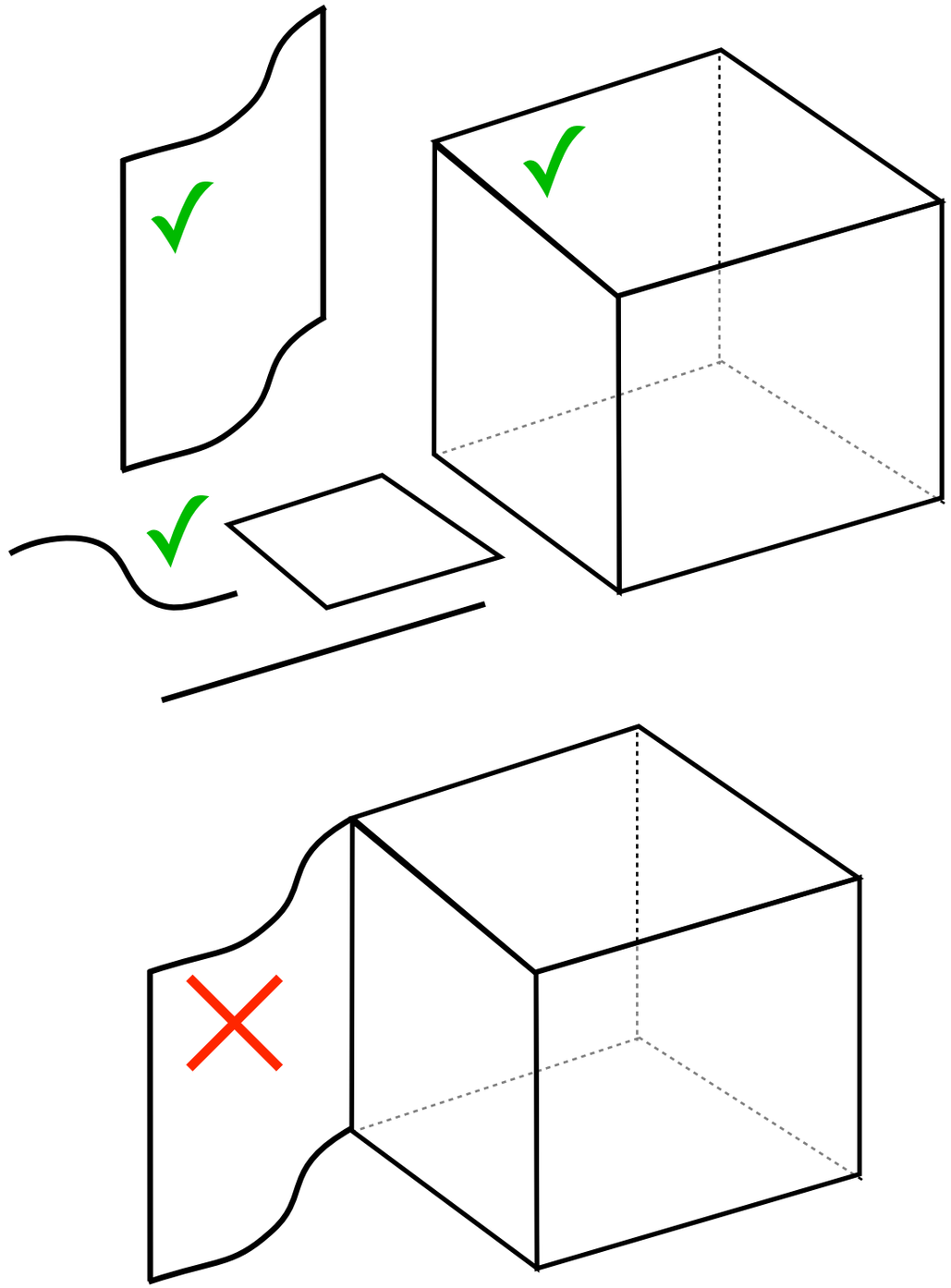
Interaction with geometry: the Triangulation class

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



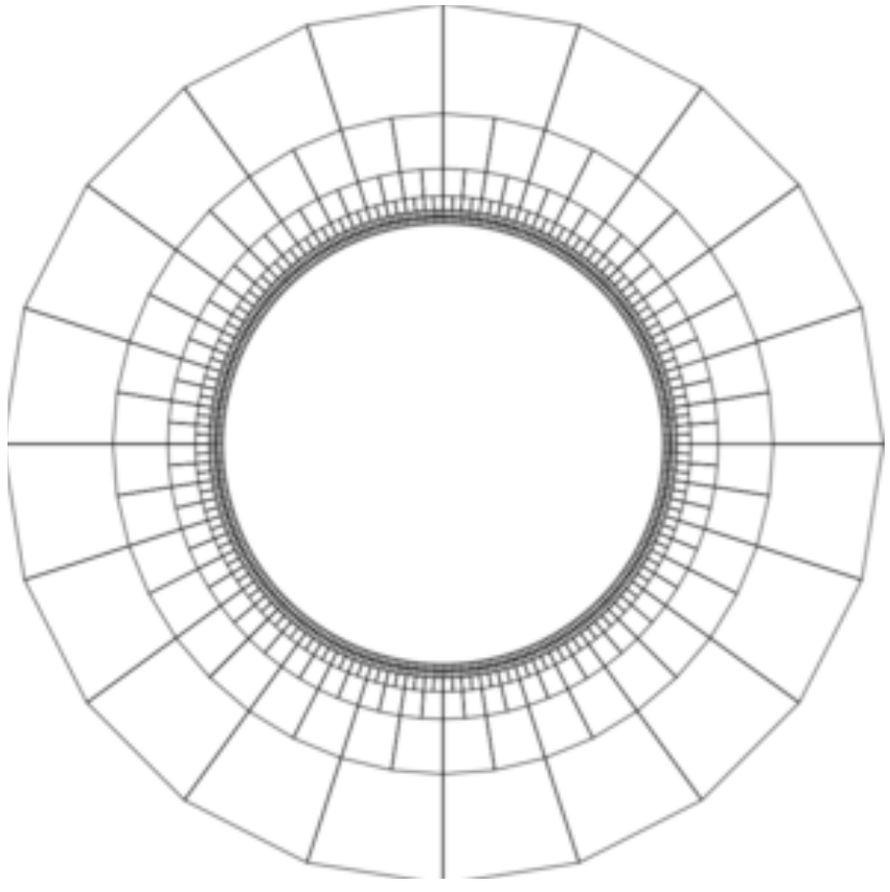
Interaction with geometry: the Triangulation class

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

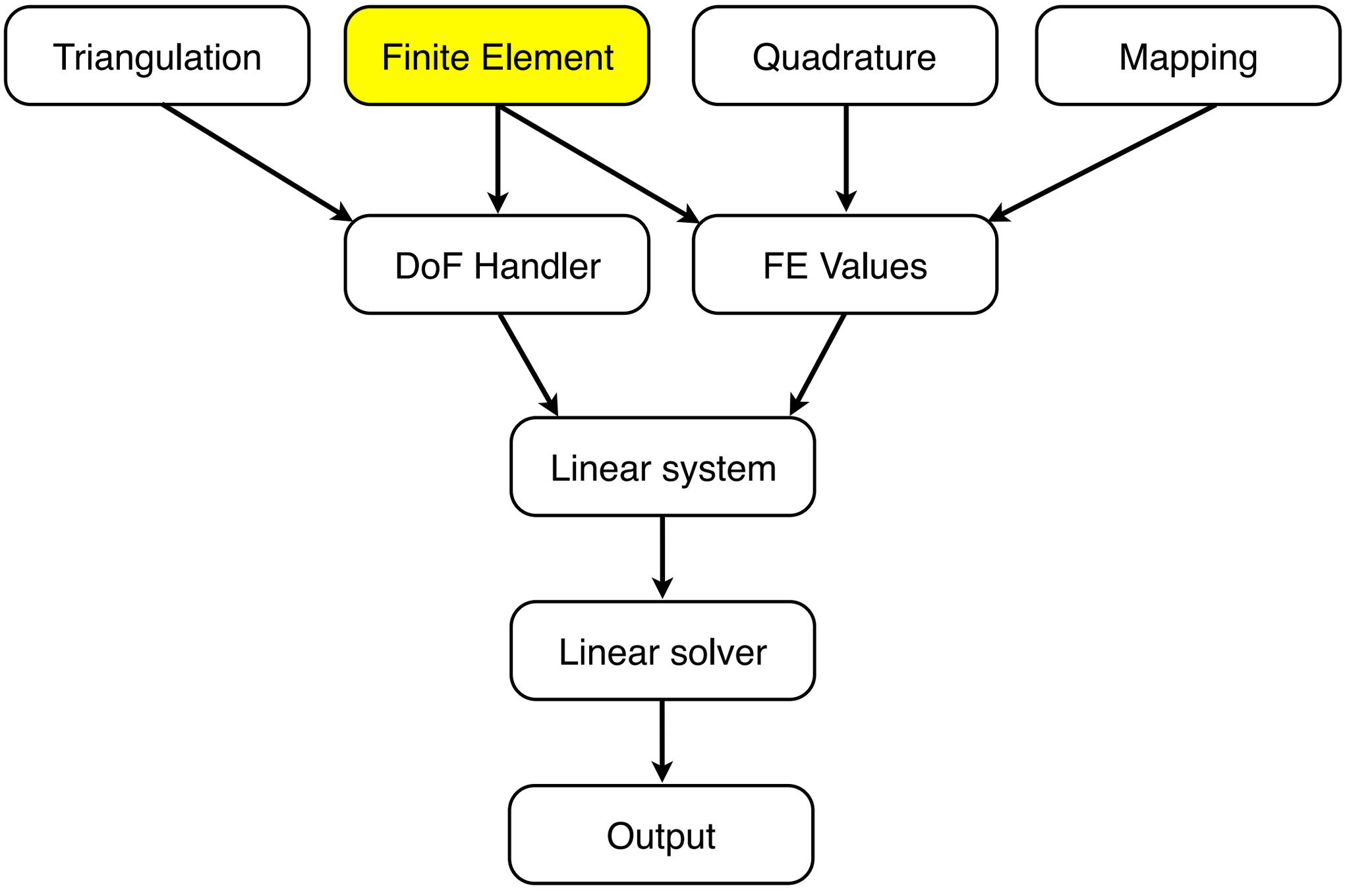


Interaction with geometry: the Triangulation class

- Demonstration: Step-1, step-49,
  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



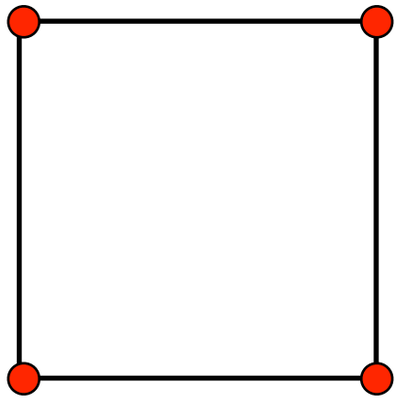
Structure of a prototypical FE problem



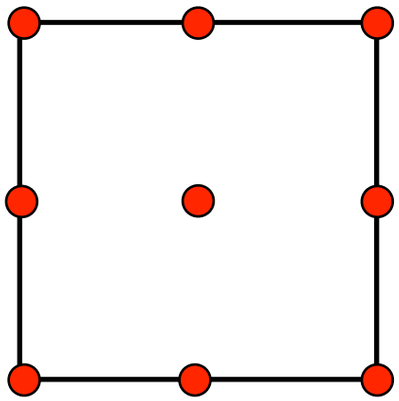
Assigning degrees-of-freedom: the FiniteElement classes

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

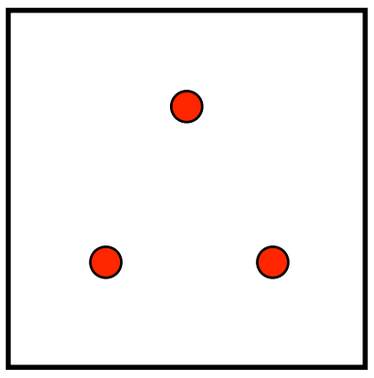
`FE_Q<2>(1)`



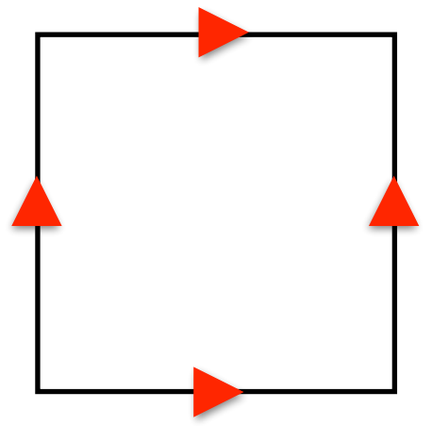
`FE_Q<2>(2)`



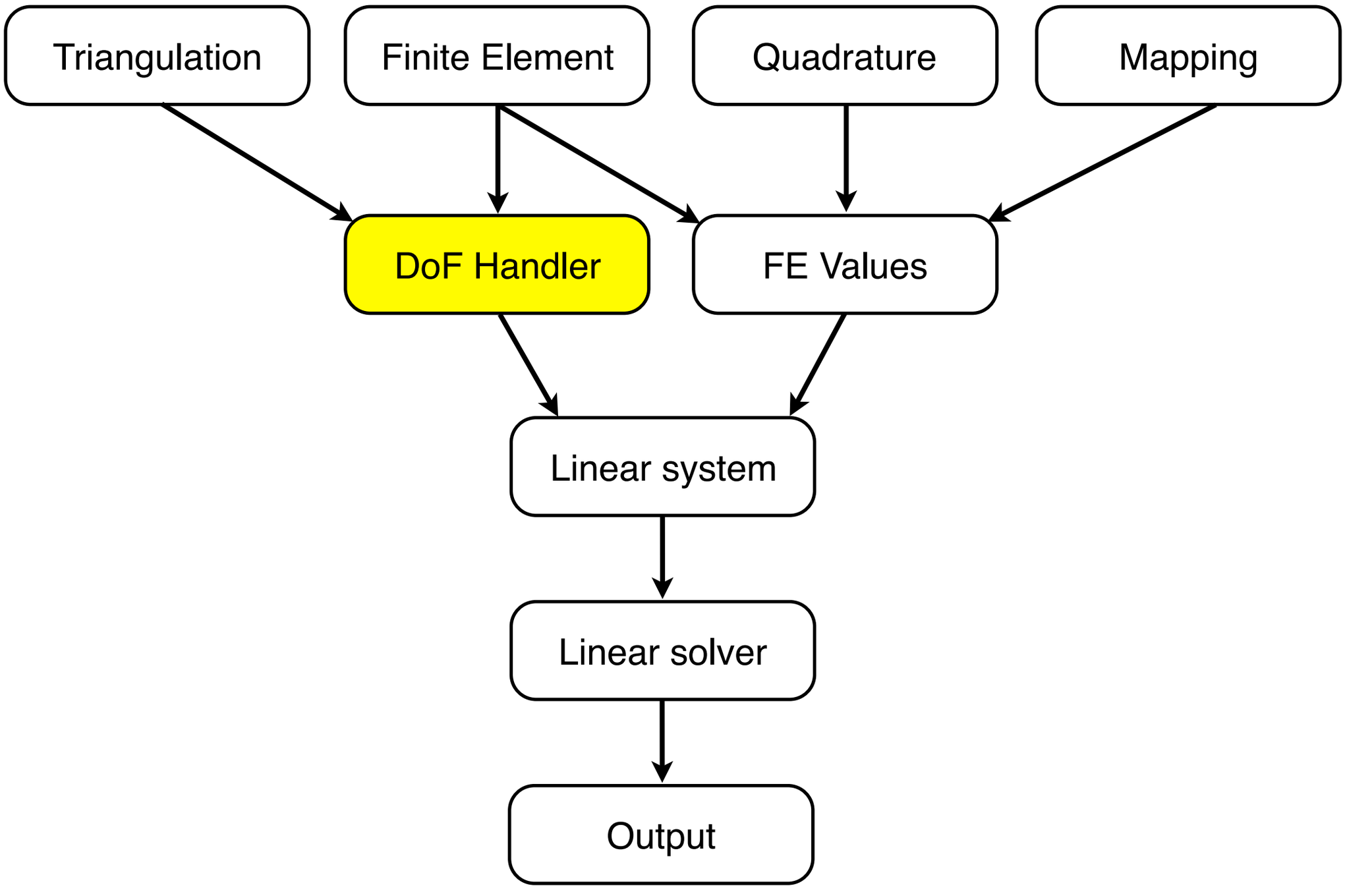
`FE_DGPMonomial<2>(1)`



`FE_Nedelec<2>(0)`

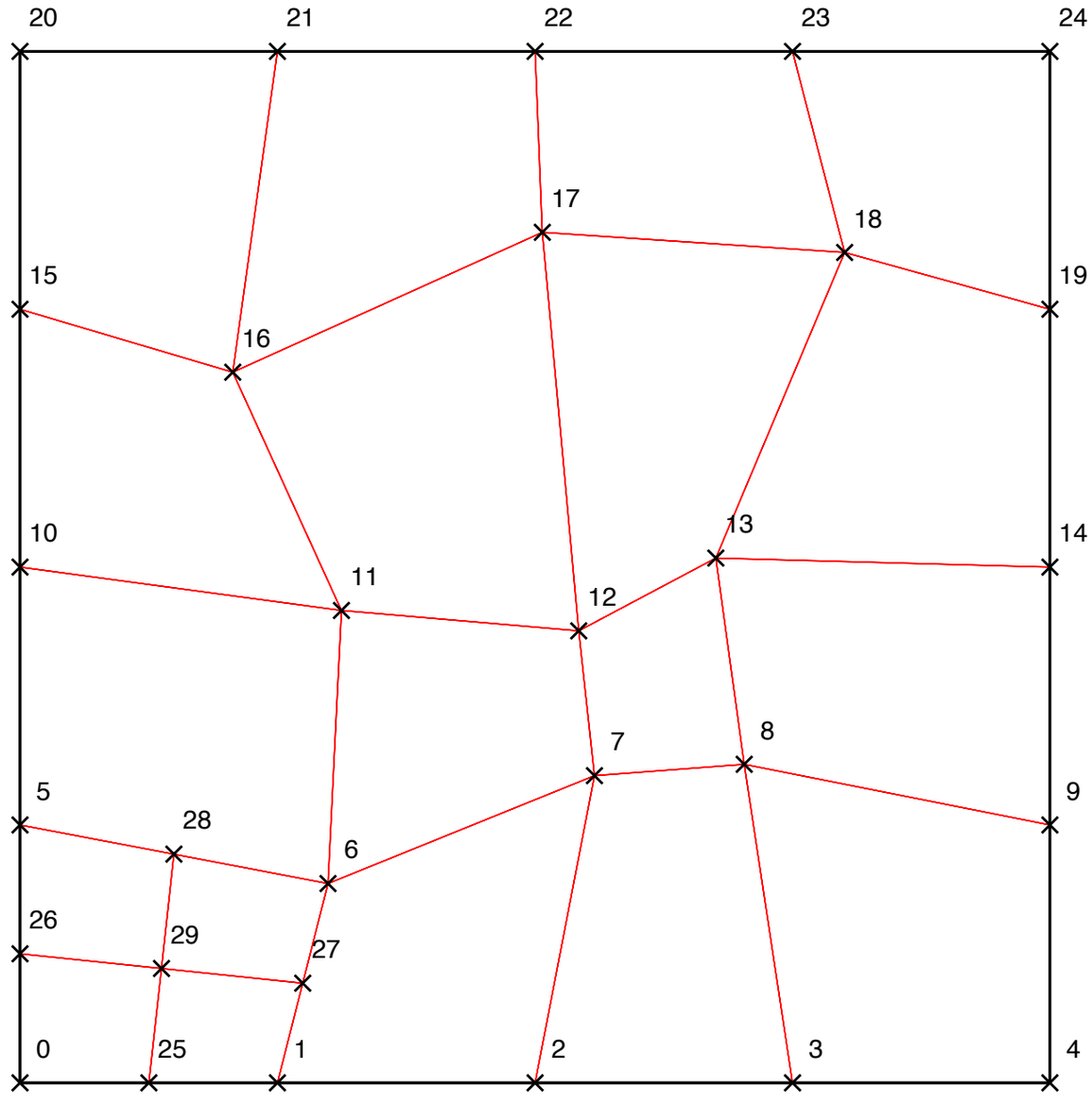


Structure of a prototypical FE problem



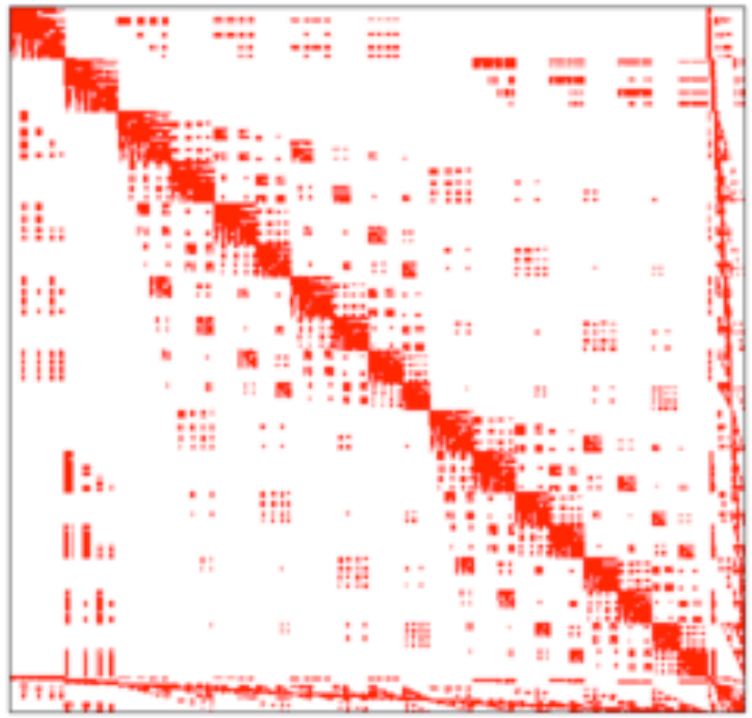
Assigning degrees-of-freedom: the DoFHandler class

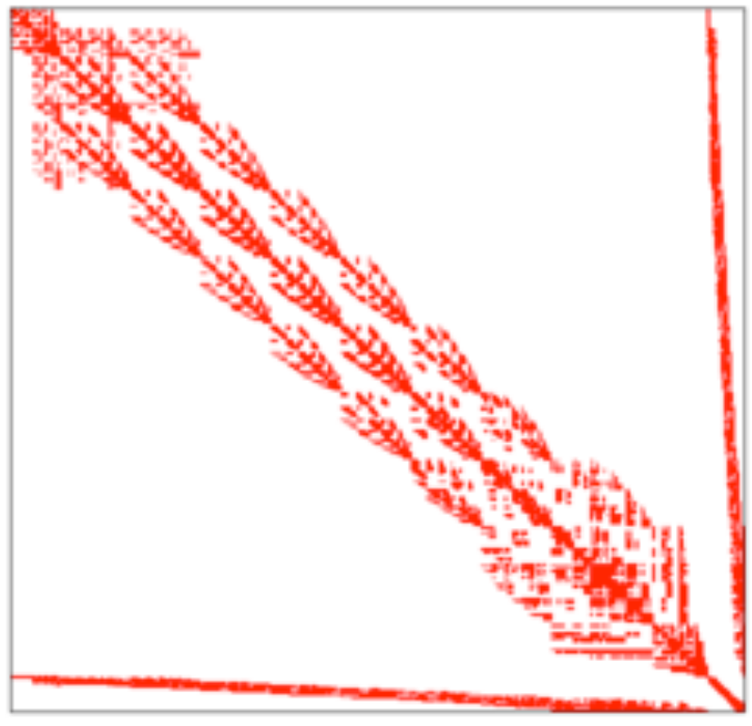
- DoFHandler assigns DoF's to grid
  - Important: separate to Triangulation!
- Unified way to access DoF's, regardless of FE used
  - e.g. Discontinuous elements: support points not necessarily at vertices
- Fast access and grid traversal
  - STL-type cell iterators
  - Access to faces and edges from cells
- Disadvantage
  - Not straight-forward (but possible) to ask location of nodes



Assigning degrees-of-freedom: the DoFRenumbering namespace

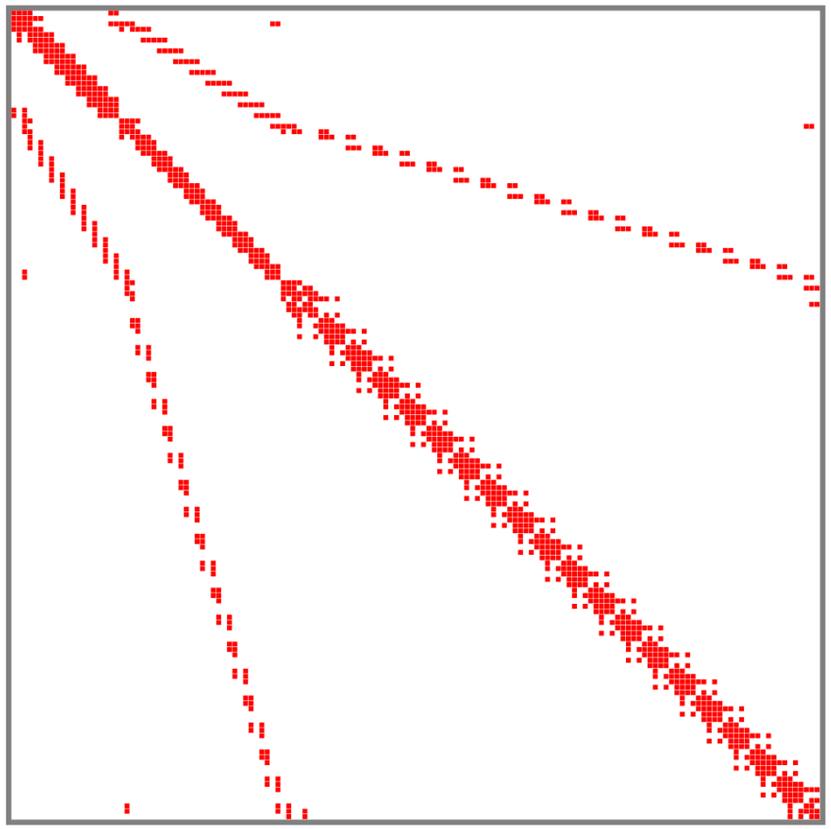
- Renumbering schemes
  - Cuthill McKee
  - King
  - Downwind
- Reduce bandwidth
- Collect like-components
- Induce block-structure
- Directional (fluid flow)
- MPI subdomain





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

- Demonstration: Step-2
  https://www.dealii.org/9.0.0/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
  - Visualizing sparsity patterns



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?

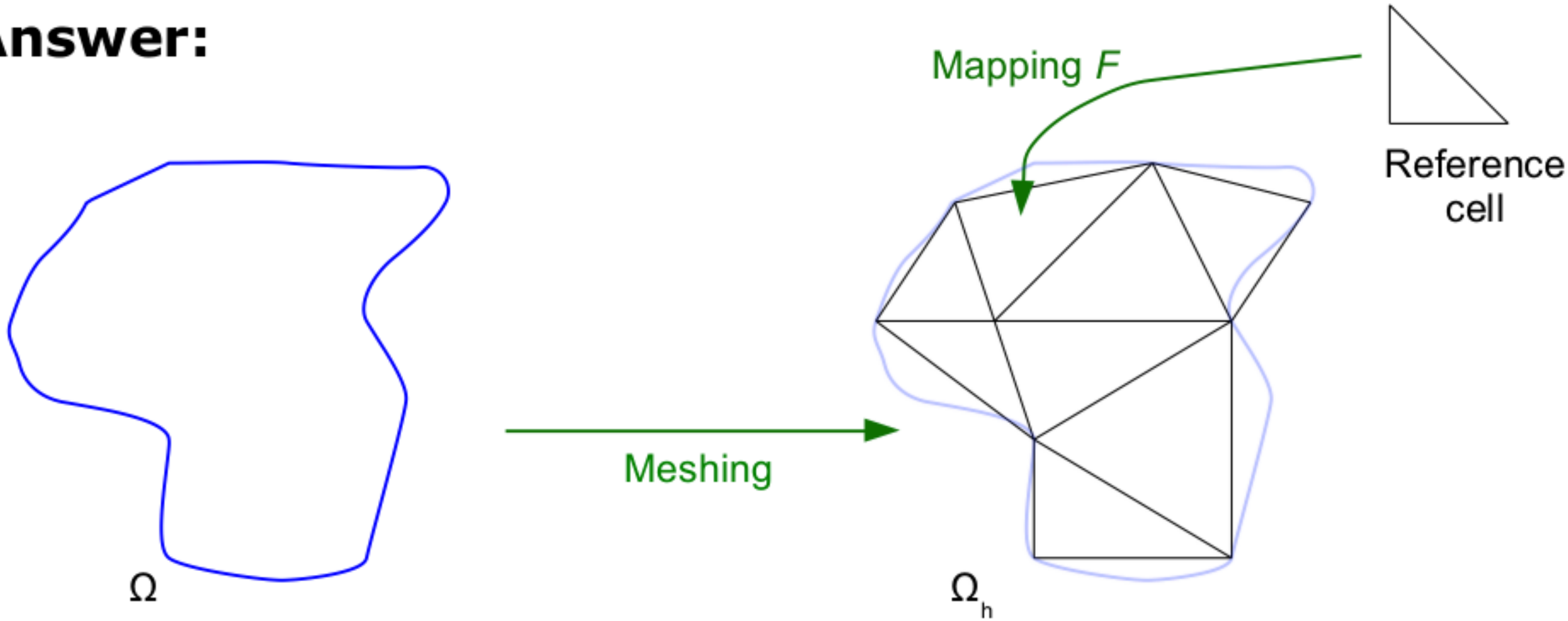
Answer: In the finite element method, this is done using the following concepts:

- 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?

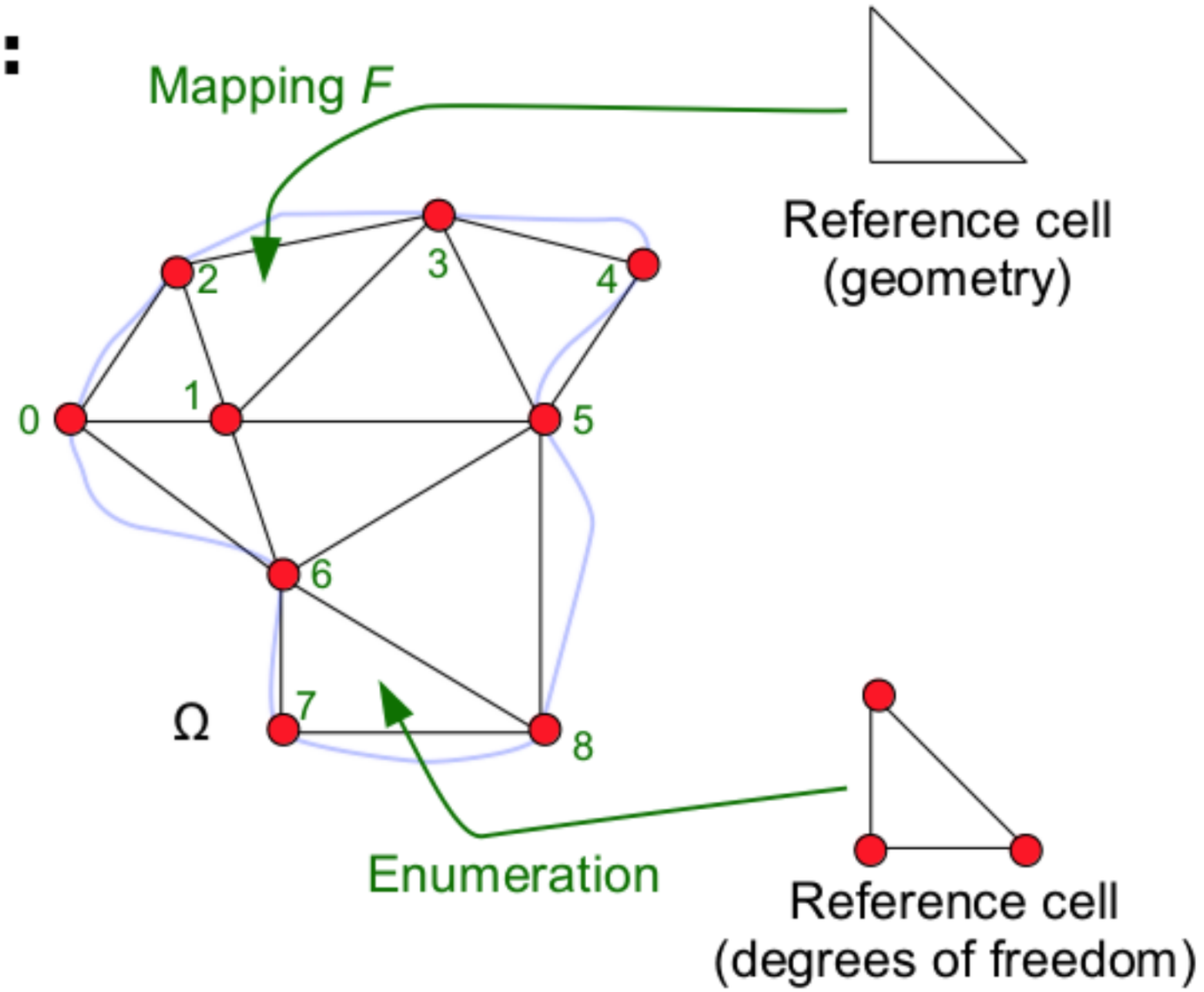
Answer:



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

$$AU = F$$

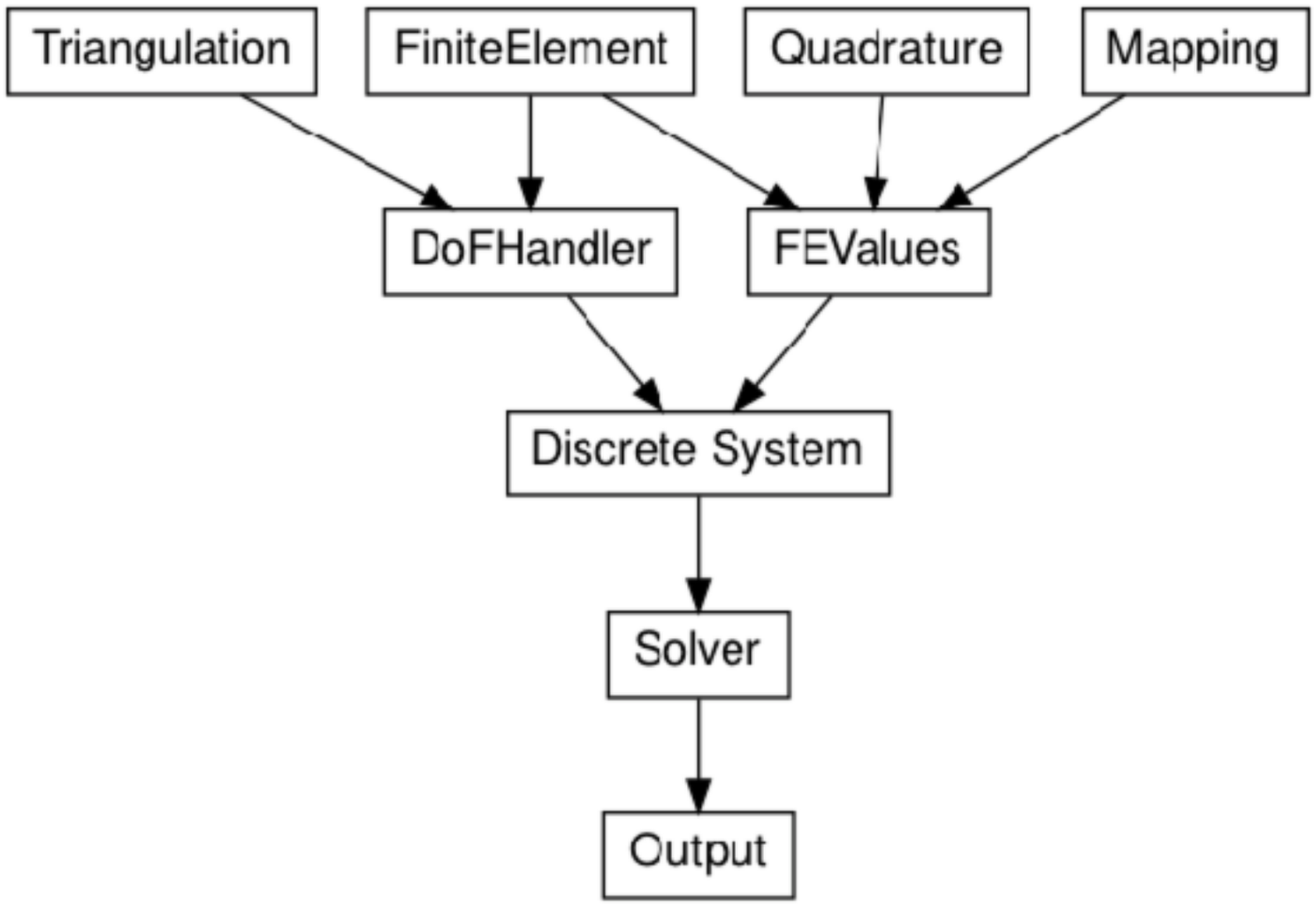
Answers: The goal is not to solve the linear system, but to do something with its solution:

- 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.*

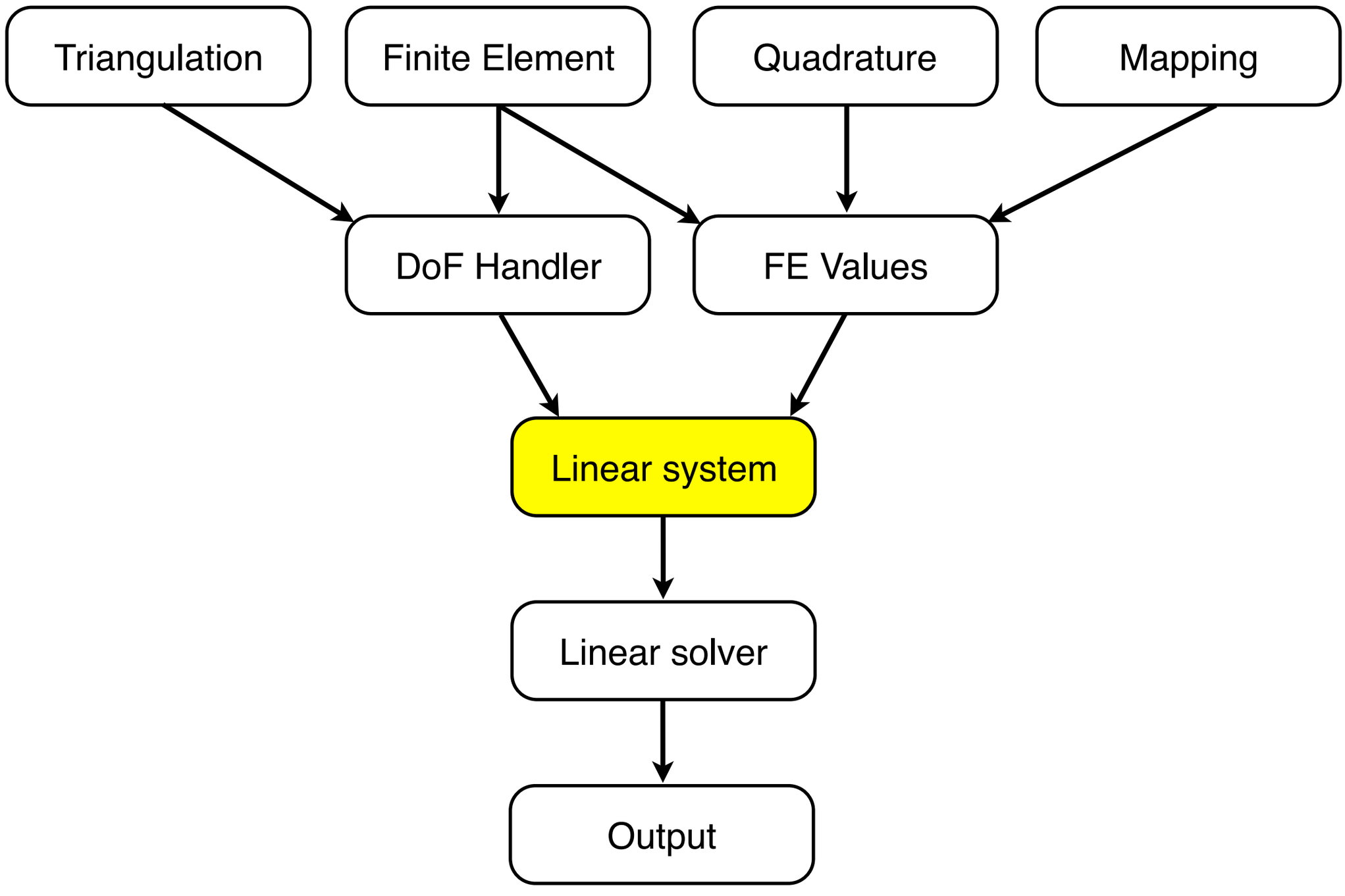
Aims for this module

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

Reference material

- Tutorials
  - Step-3
    https://dealii.org/9.0.0/doxygen/deal.II/step_3.html
- Documentation
  - https://www.dealii.org/9.0.0/doxygen/deal.II/group__FE__vs__Mapping__vs__FEValues.html
  - https://www.dealii.org/9.0.0/doxygen/deal.II/group__UpdateFlags.html

Structure of a prototypical FE problem



Sparse linear systems

- Minimize data storage
  - Evaluate grid connectivity
- Functions to help set up
  - Sparsity pattern
  - 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 interpolatory FEM
- Neumann boundary conditions
  - Implementation dependent
- Other constraints need special consideration
  - Periodic boundary conditions
  - Refinement with hanging nodes
  - Some time-dependent formulations

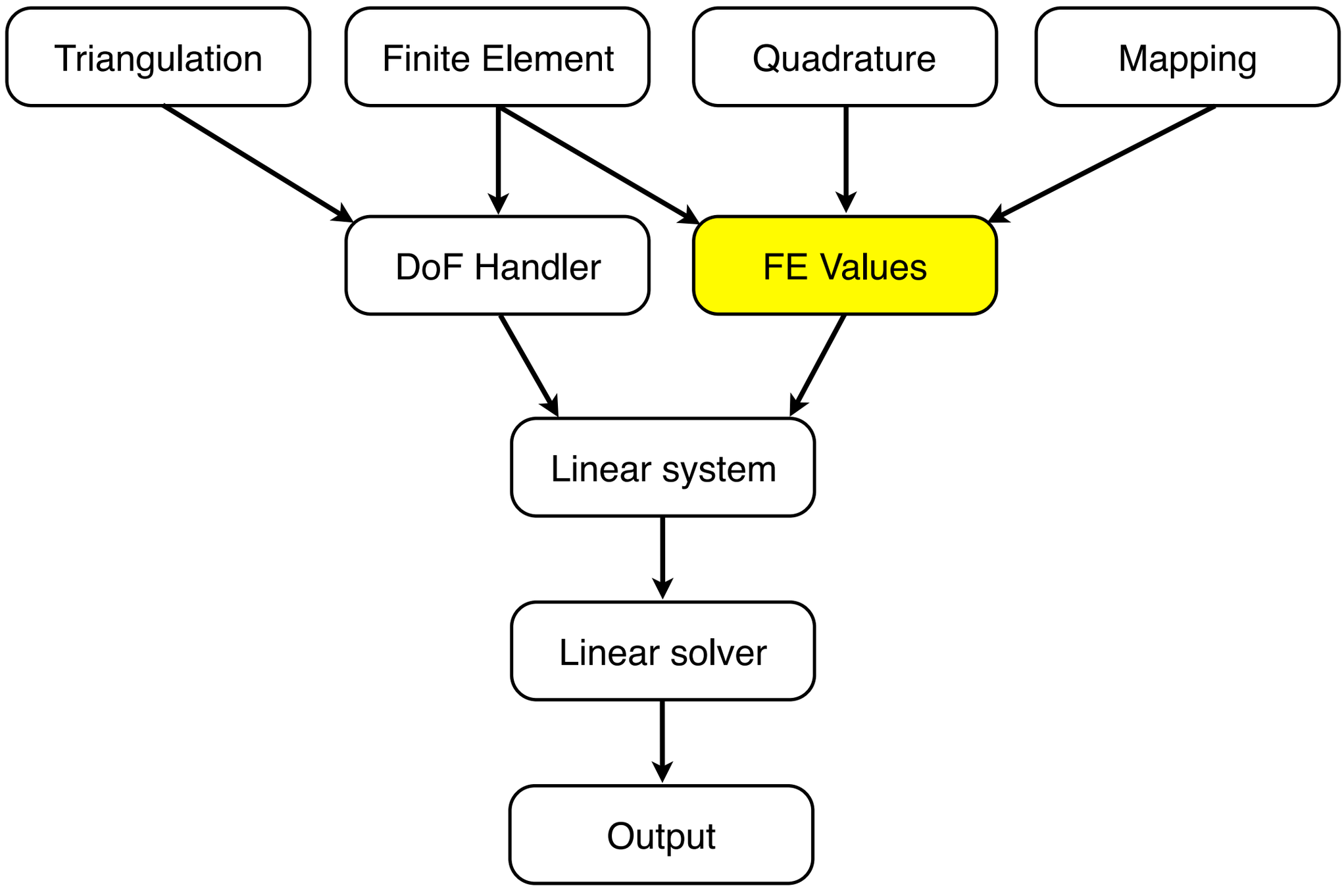
$$[K]\{d\} = \{F\}$$

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

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

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

Structure of a prototypical FE problem



Integration on a cell: the FEValues class

$$K = \int_\Omega \nabla \delta\phi(\mathbf{x}) \cdot k \nabla \phi(\mathbf{x}) dV$$

$$\approx \delta\phi^I \sum_K \left(\int_{\Omega_K^h} \nabla N^I(\mathbf{x}) \cdot k \nabla N^J(\mathbf{x}) dV^h \right) \phi^J \qquad J_K = \frac{\partial \mathbf{X}^{\boldsymbol{\xi}}}{\partial \mathbf{X}}$$

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

$$\approx \delta\phi^I \underbrace{\sum_K \left(\sum_q J_K^{-1}(\hat{\mathbf{x}}_q) \, \hat{\nabla} \hat{N}^I(\hat{\mathbf{x}}_q) \cdot k_q \, J_K^{-1}(\hat{\mathbf{x}}_q) \, \hat{\nabla} \hat{N}^J(\hat{\mathbf{x}}_q) \, |\det J_K(\hat{\mathbf{x}}_q)| \, w_q \right)}_{K_{IJ}} \phi^J$$

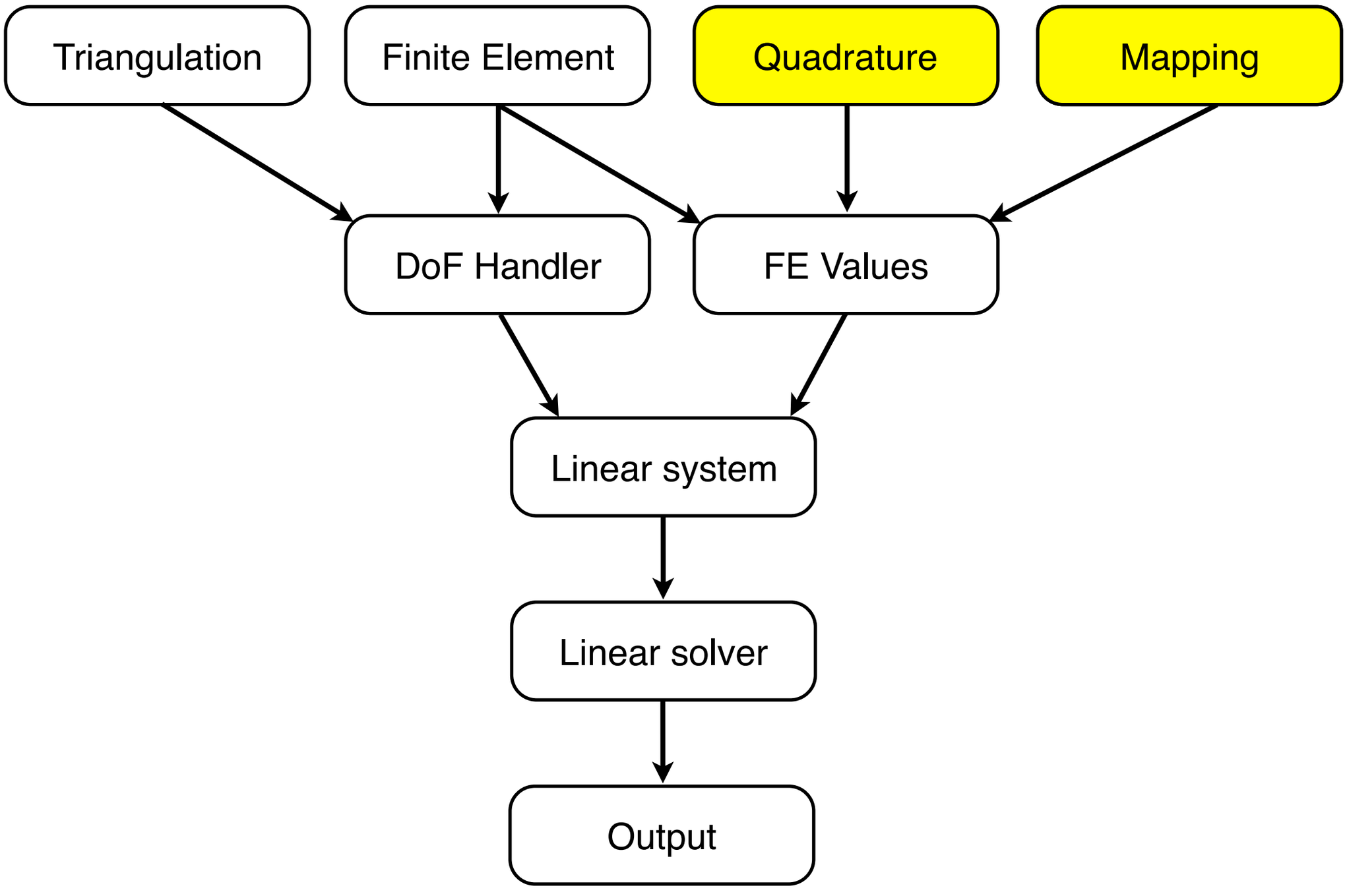
Integration on a cell: the FEValues class

- 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 optimizations

$$K_{IJ} = \sum_K \left(\sum_q J_K^{-1}(\hat{\mathbf{x}}_q)\,\hat{\nabla}\hat{N}^I(\hat{\mathbf{x}}_q) \cdot J_K^{-1}(\hat{\mathbf{x}}_q)\,\hat{\nabla}\hat{N}^J(\hat{\mathbf{x}}_q)\,|\det J_K(\hat{\mathbf{x}}_q)|\,w_q \right)$$

```
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



Matrix form

$$\mathbf{K} \cdot \mathbf{u} = \mathbf{F}$$

$$K_{ij} := a(N_i, N_j) \qquad i, j \in \mathcal{N}_U$$

$$F_i := (N_i, f) + (N_i, h)_{\partial\Omega} - \sum_{j \in \mathcal{N}_D} a(N_i, N_j) q(\mathbf{x}_j)$$

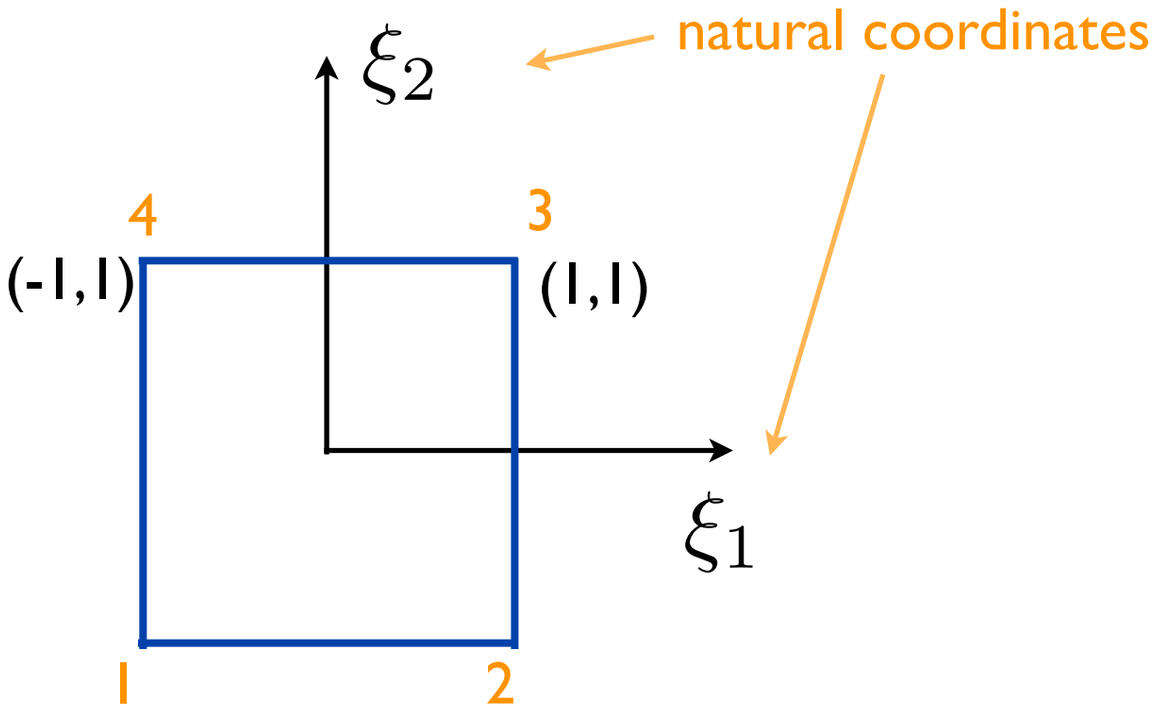
$$(S) = (W) \approx (W^h) = (D)$$

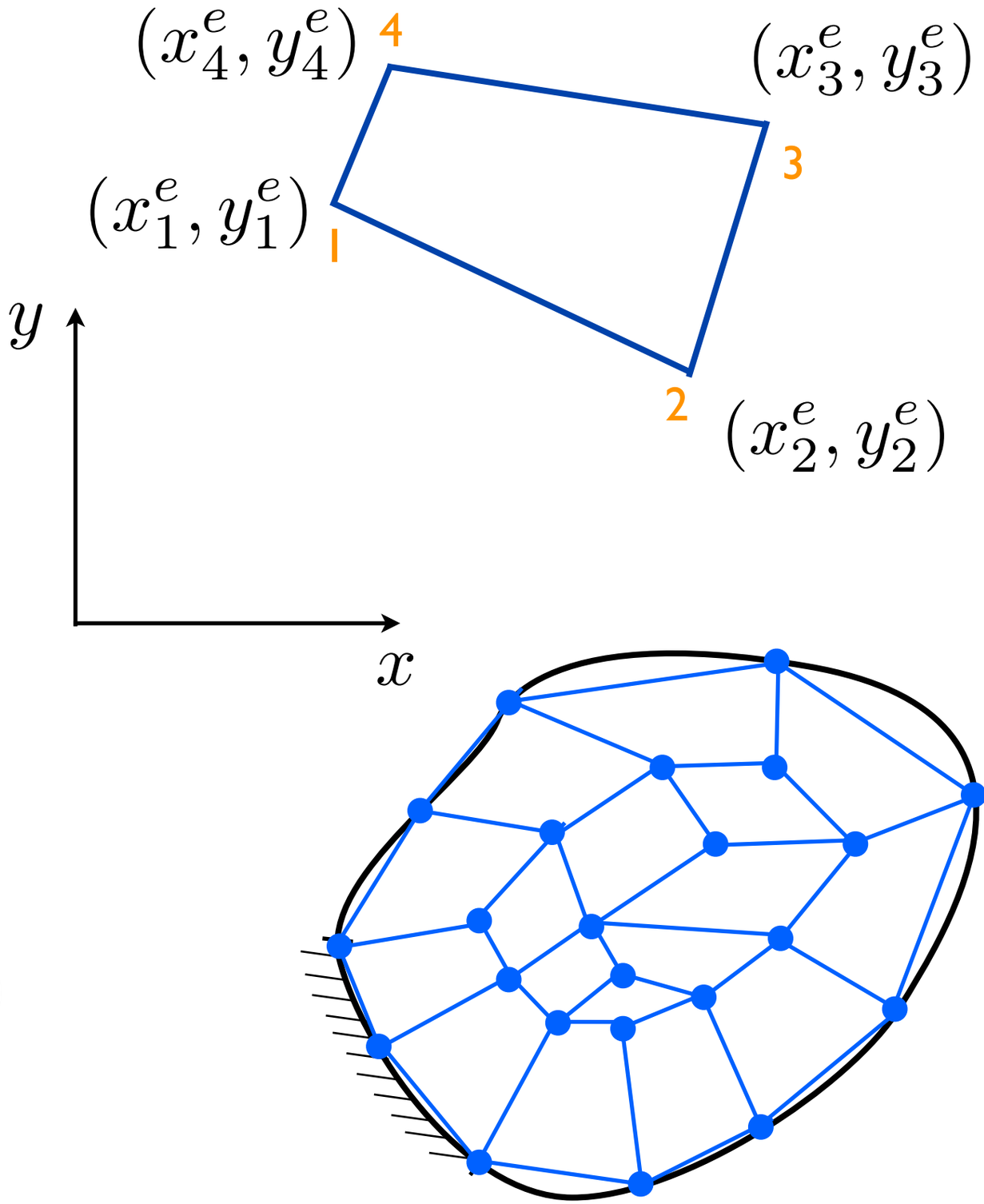
need to evaluate integrals numerically

$$a(N_i, N_j) := \sum_K \int_{\Omega_K} \nabla N_i \cdot \mathbf{k} \cdot \nabla N_j \mathrm{d}v$$

$$(N_i, f) := \sum_K \int_{\Omega_K} N_i f(\mathbf{x}) \mathrm{d}v \qquad (w, h)_{\partial\Omega} := \sum_K \int_{\partial\Omega_K^N} wh \mathrm{d}s$$

Q1 mapping





we can construct the mapping between the two elements

$$\boldsymbol{x} = \boldsymbol{x}(\boldsymbol{\xi})$$

Bilinear Quadrilateral Element

Bilinear expansion

$$x(\xi_1, \xi_2) =: \alpha_0 + \alpha_1\xi_1 + \alpha_2\xi_2 + \alpha_3\xi_1\xi_2$$
$$y(\xi_1, \xi_2) =: \beta_0 + \beta_1\xi_1 + \beta_2\xi_2 + \beta_3\xi_1\xi_2$$

+

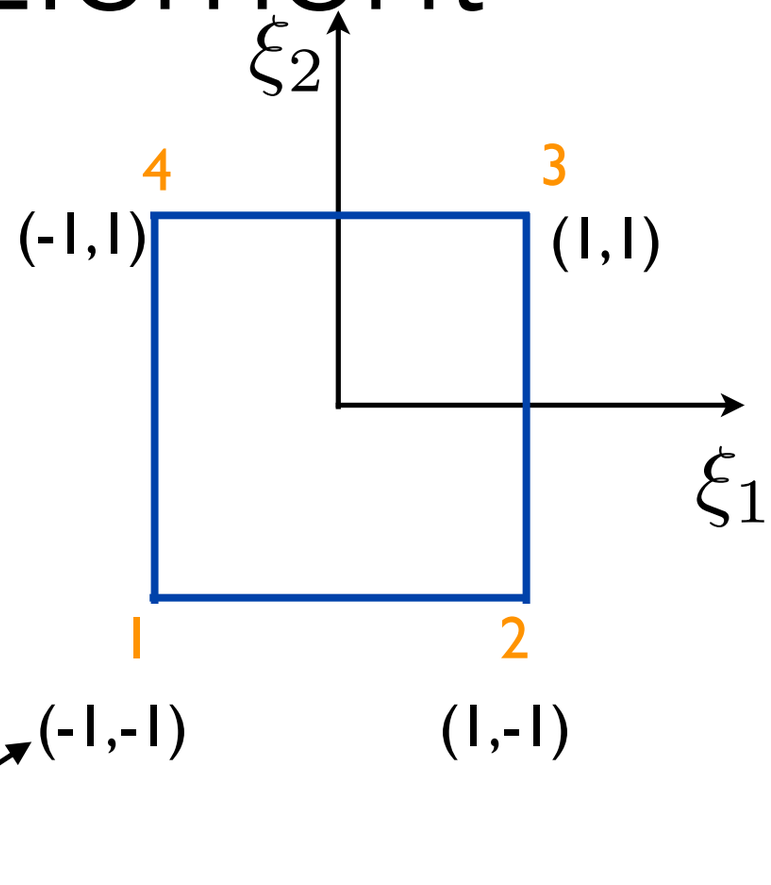
$$x(\xi_1^a, \xi_2^a) = x_a^e \qquad a = \overline{1,4}$$
$$y(\xi_1^a, \xi_2^a) = y_a^e$$

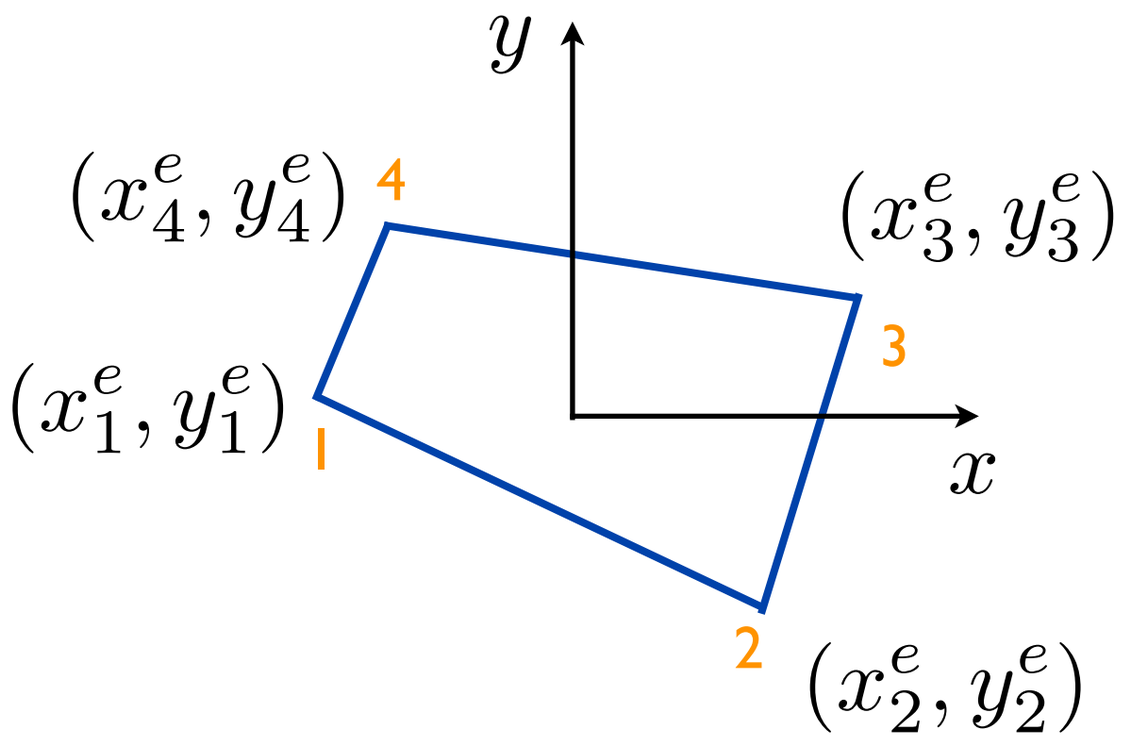
=

$$\boldsymbol{x}(\boldsymbol{\xi}) = \sum_{a=1}^{4} N_a(\boldsymbol{\xi})\boldsymbol{x}_a^e$$

maps any point in the reference element to the actual element

$$N_a(\boldsymbol{\xi}) = \frac{1}{4}[1 + \xi_1^a\xi_1][1 + \xi_2^a\xi_2]$$





Mapping to the reference element:

$$\mathbf{J} := \frac{\partial \mathbf{x}}{\partial \xi}$$

$$\nabla = \frac{\partial}{\partial x_i}\mathbf{e}_i$$

$$\mathrm{d}v = \det(\mathbf{J}_K)\mathrm{d}\,\hat{v}$$

$$\mathrm{grad}(\bullet) = (\bullet)\nabla = \frac{\partial(\bullet)}{\partial x_i}\mathbf{e}_i = \frac{\partial(\bullet)}{\partial \xi_j}\frac{\partial \xi_j}{\partial x_i}\mathbf{e}_i = \widehat{\mathrm{grad}}\,(\bullet)\cdot\mathbf{J}_K^{-1}$$

$$(S) = (W) \approx (W^h) = (D) \approx (D^q)$$

$$a(N_i, N_j) = \sum_K \int_{\Omega_K} \mathrm{grad}\, N_i(\mathbf{x}) \cdot \mathrm{grad}\, N_j(\mathbf{x})\mathrm{d}v$$

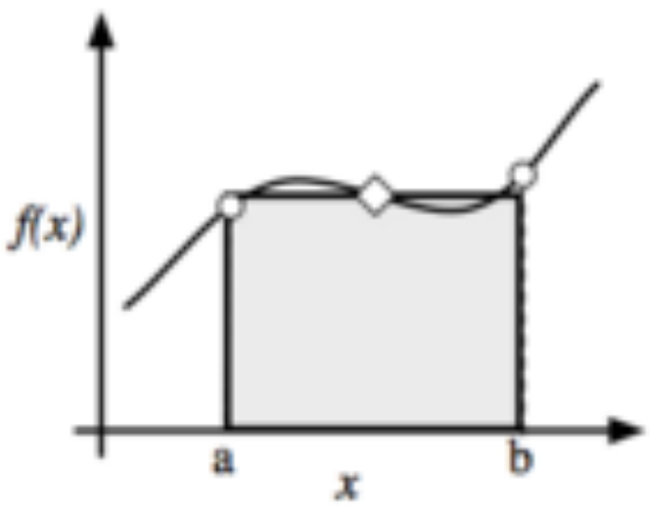
$$= \sum_K \int_{\hat{\Omega}_K} [\widehat{\mathrm{grad}}\ \widehat{N}_i(\xi)\cdot\mathbf{J}_K^{-1}(\xi)]\cdot[\widehat{\mathrm{grad}}\ \widehat{N}_j(\xi)\cdot\mathbf{J}_K^{-1}(\xi)]\det(\mathbf{J}_K(\xi))\mathrm{d}\,\hat{v}$$

$$\approx \sum_K \sum_q [\widehat{\mathrm{grad}}\ \widehat{N}_i(\xi_q)\cdot\mathbf{J}_K^{-1}(\xi_q)]\cdot[\widehat{\mathrm{grad}}\ \widehat{N}_j(\xi_q)\cdot\mathbf{J}_K^{-1}(\xi_q)]\det(\mathbf{J}_K(\xi_q))w_q$$

do not depend on a particular cell

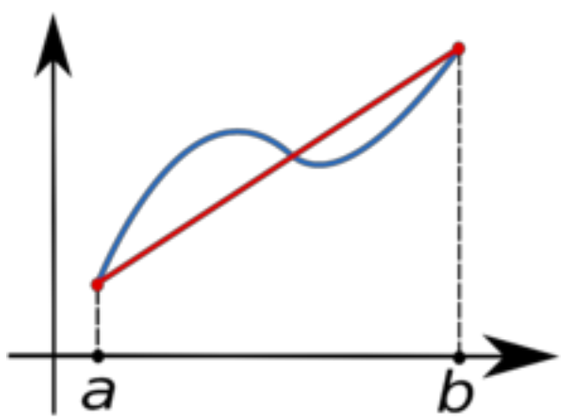
Integration rules:

1. midpoint



$$\int_a^b f(x)\mathrm{d}x \approx f\left(\frac{a+b}{2}\right)\ [b-a]$$

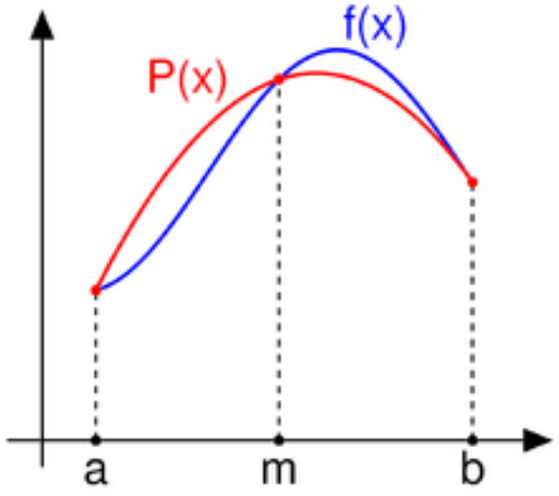
2. trapezoidal



$$\int_a^b f(x)\mathrm{d}x \approx \left[\frac{f(a)+f(b)}{2}\right]\ [b-a]$$

Integration rules:

3. Simpson



$$\int_a^b f(x)\mathrm{d}x \approx \left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \frac{b-a}{6}$$

4. Gauss quadrature rule

Constructed to be exact for polynomials of degree 2n-1

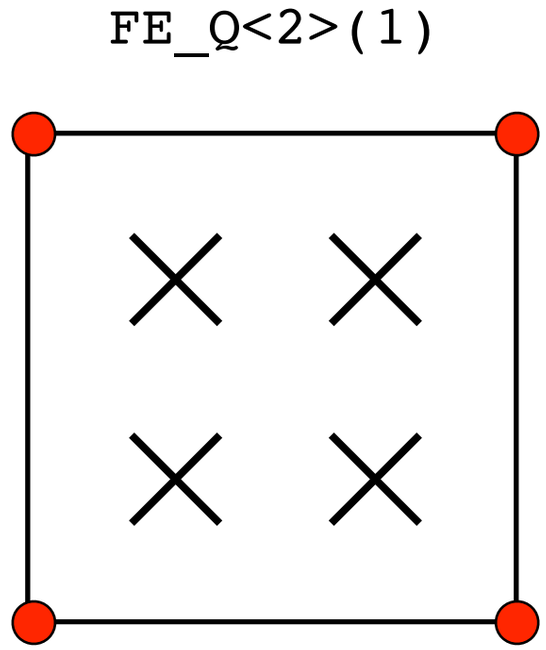
$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_q f(x_q)w_q$$

n_q	x_1	x_2	x_3	w_1	w_2	w_3
1	0			2		
2	$-1/\sqrt{3}$	$1/\sqrt{3}$		1	1	
3	$-\sqrt{3/5}$	0	$\sqrt{3/5}$	$5/9$	$8/9$	$5/9$

there are other integration rules: Monte Carlo, Newton-Cotes, Runge-Kutta,...

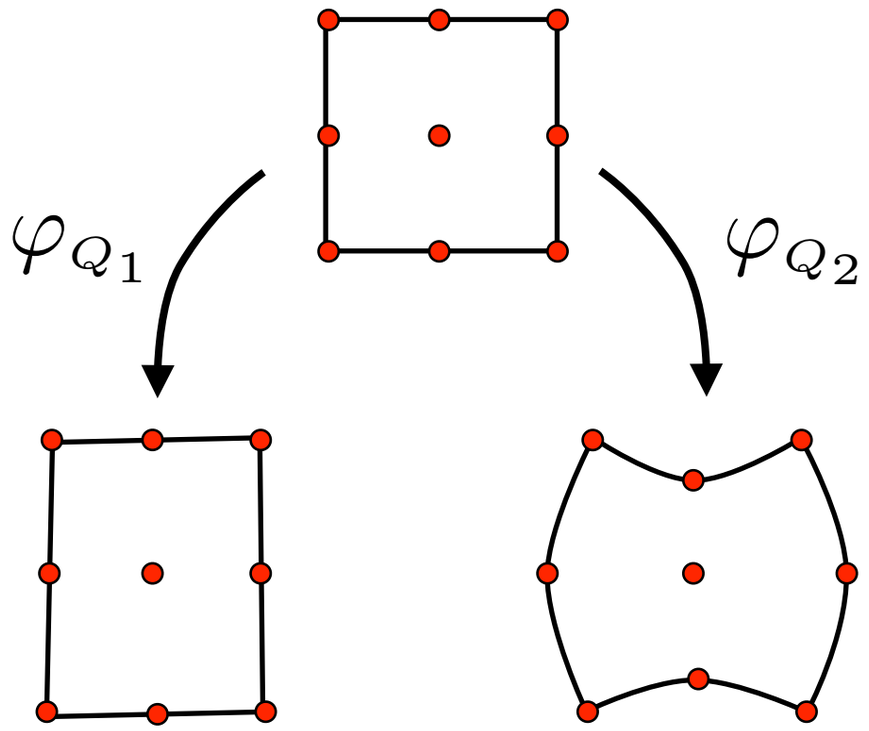
Integration on a cell: the Quadrature classes

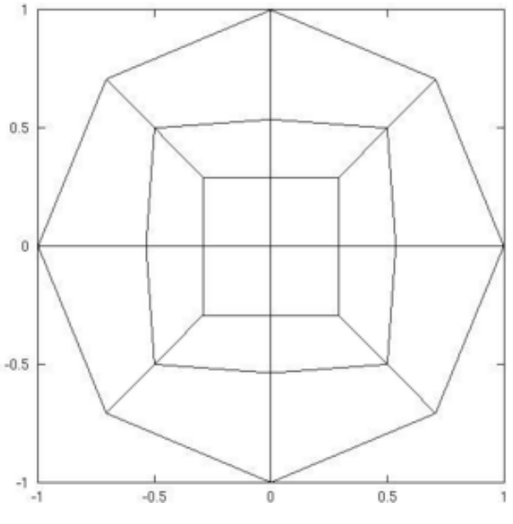
- QGauss<dim> n-Order Gauss quadrature
- Other rules
  - QGaussLobattom<dim> Gauss Lobatto
  - QSimpson<dim> Simpson
  - QTrapez<dim> Trapezoidal
  - QMidpoint Midpoint
  - …

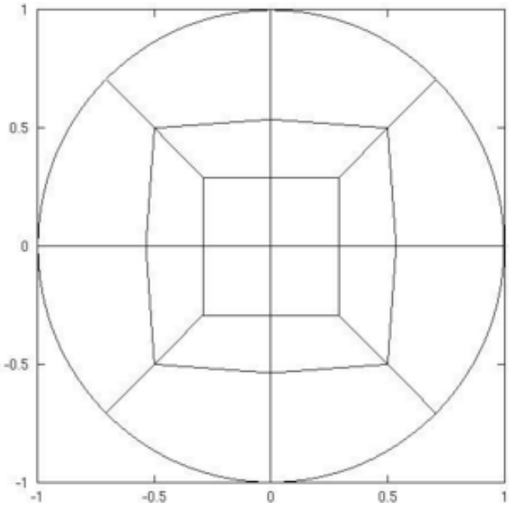


Integration on a cell: the Mapping classes

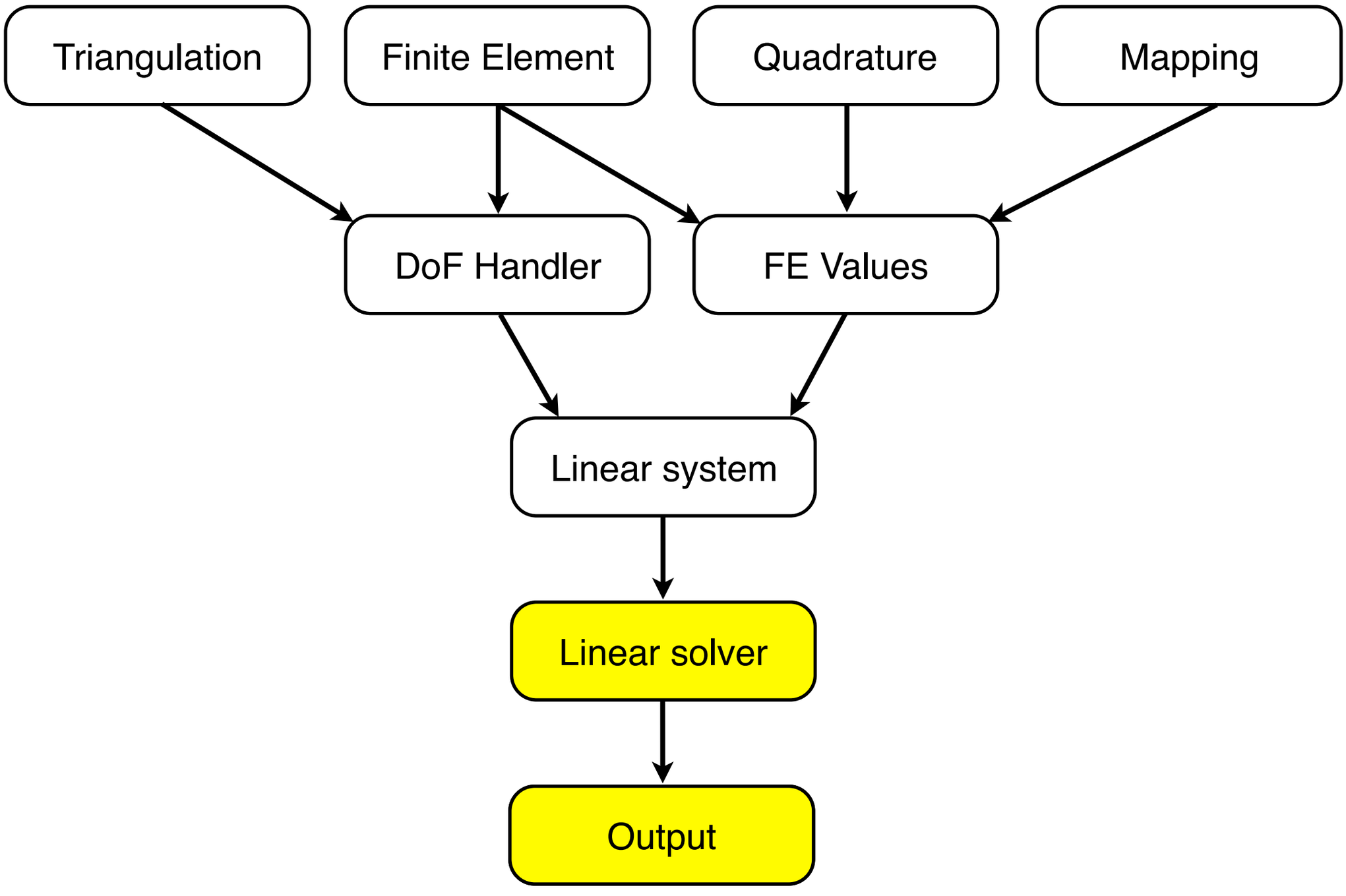
- n-order mappings
  - Increase accuracy of:
    - Integration schemes
    - Surface basis vectors
- Lagrangian / Eulerian
  - Latter useful for fluid and contact problems, data visualization
- Boundary and interior manifolds





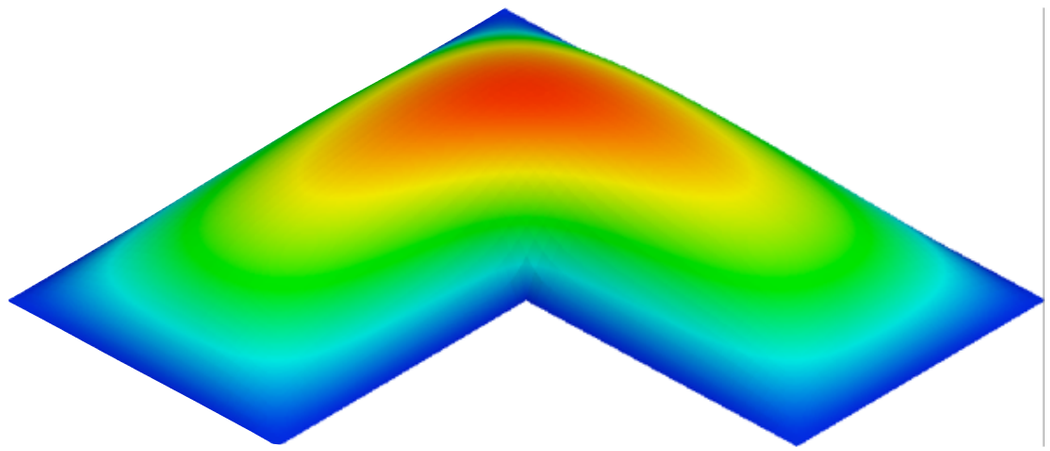


Structure of a prototypical FE problem



Solving Poisson's equation

- Demonstration: Step-3
  https://www.dealii.org/9.0.0/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



Shared memory parallelization

Denis Davydov (denis.davydov@fau.de)
Luca Heltai (luca.heltai@sissa.it)

25 February - 1 March 2019

Aims for this module

- Identify parts / blocks of code that are (easily) parallelizable
- Learn how to parallelize using
  - ThreadGroup (Posix threads)
  - Workstream (Threaded building blocks)

Reference material

- Tutorials
  - https://dealii.org/9.0.0/doxygen/deal.II/step_9.html
  - https://dealii.org/9.0.0/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/9.0.0/doxygen/deal.II/group__threads.html
  - https://www.dealii.org/9.0.0/doxygen/deal.II/namespaceWorkStream.html
  - https://dealii.org/9.0.0/doxygen/deal.II/namespaceparallel.html

Identifying parallelizable 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 parallelizable code

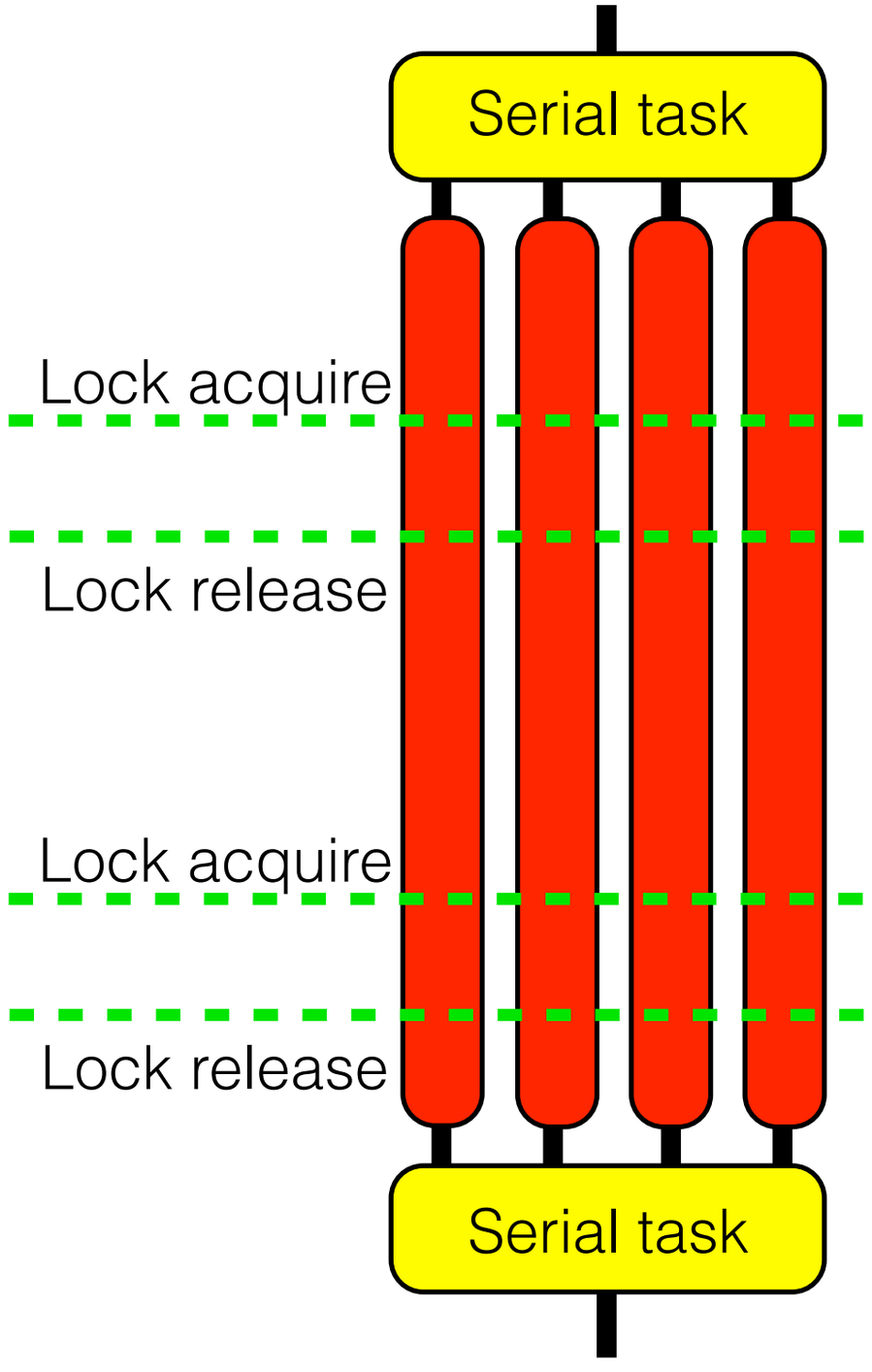
- “Embarrassingly parallelizable tasks”

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

- Many more cells than machine cores
- Computations of local matrices/vectors are mutually independent
- **Accumulation into global system matrix/vector 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 ();}
}
```

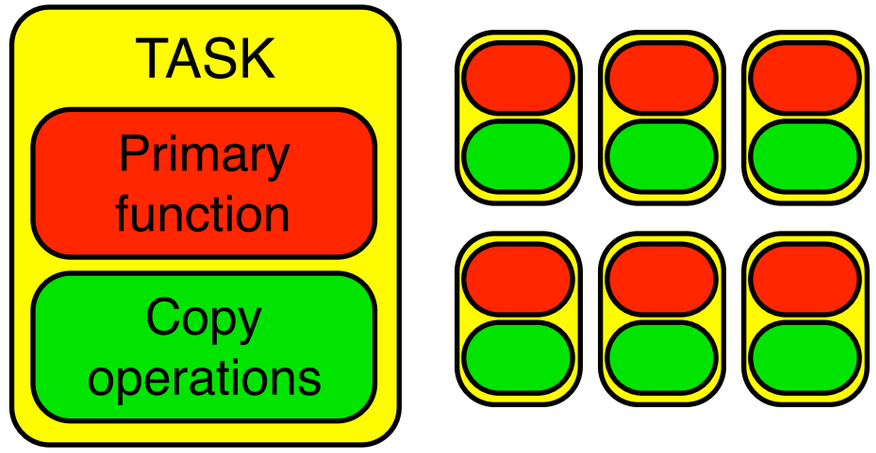
Independent threaded tasks

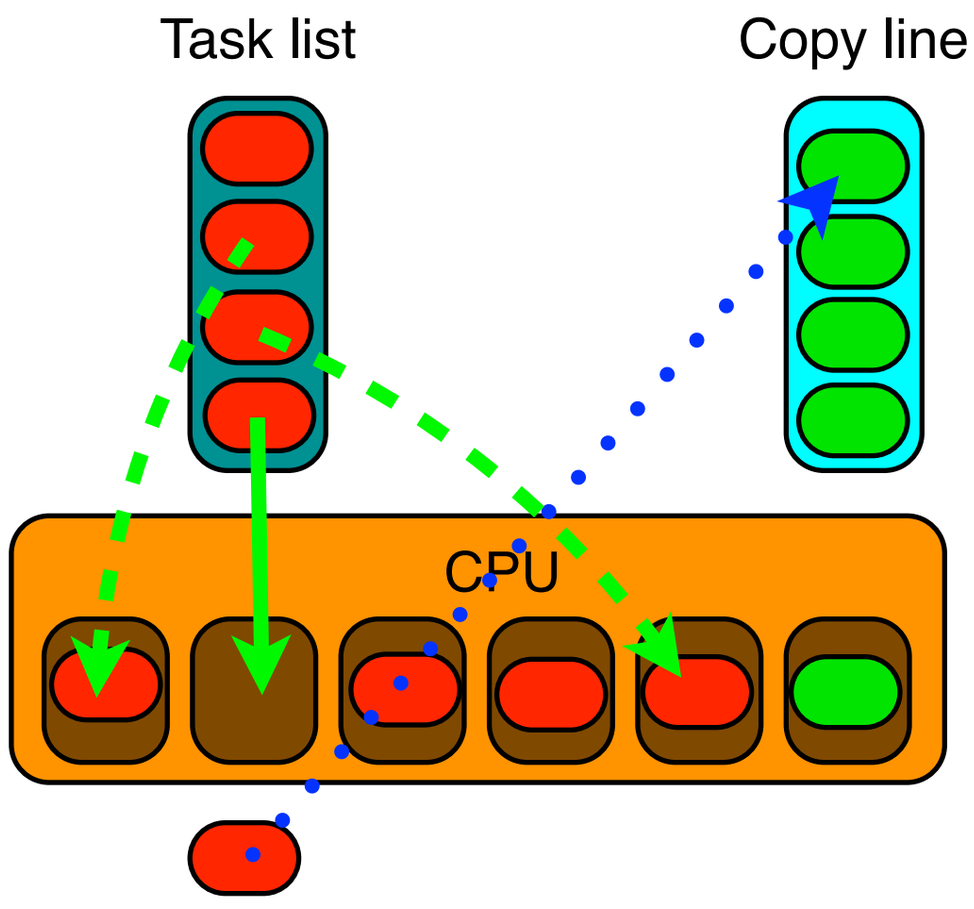
- How do we prevent data races?

```
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
- Optimizations:
  - “Automatic” load balancing
  - Overhead reduction: Works on data chunks





Creating independent threaded tasks: parallelization 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 ScratchData 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

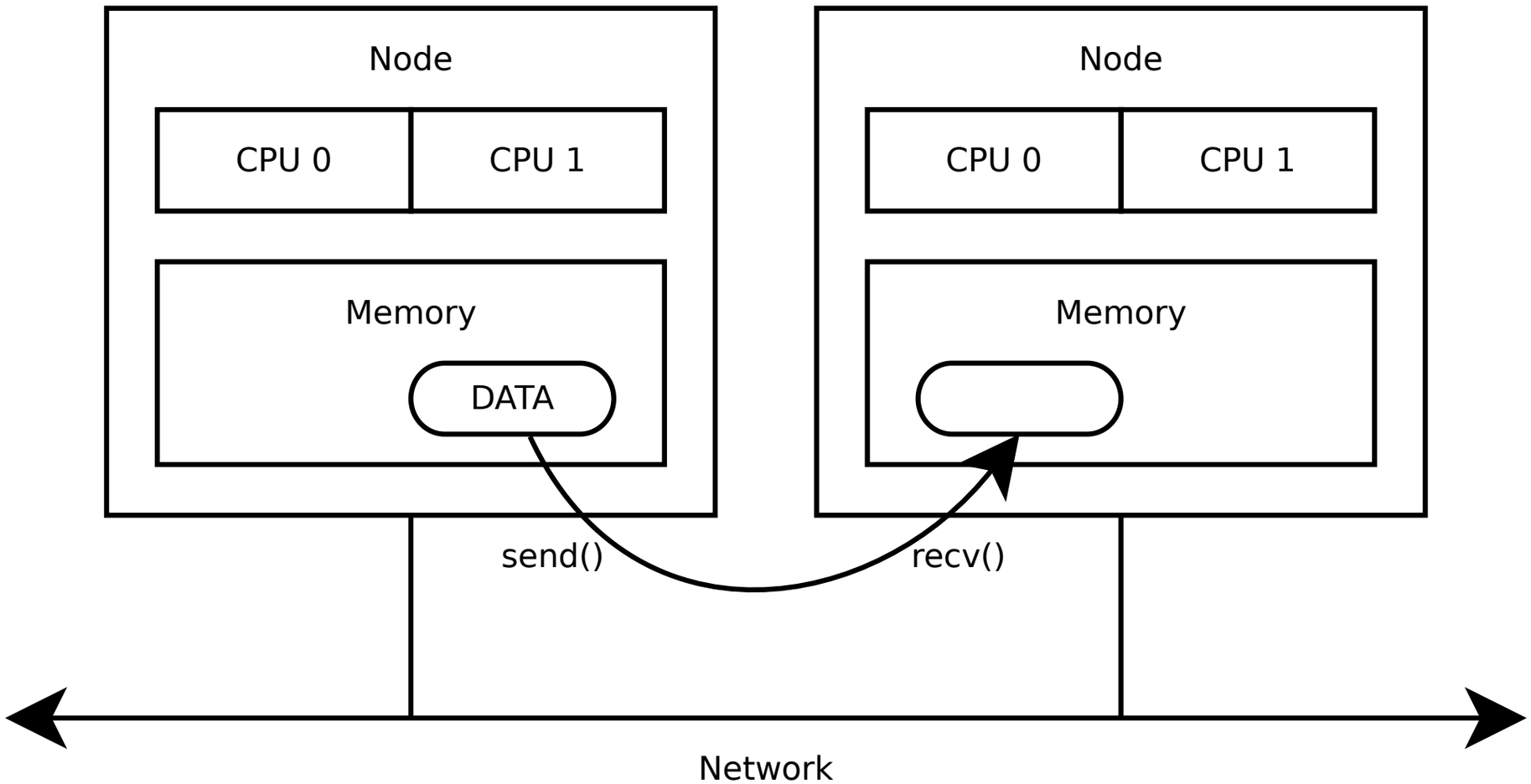
```
WorkStream::run ( dof_handler.begin_active(),
                  dof_handler.end(),
                  std::bind(&Solid::assemble_residual_one_cell,
                            this,
                            _1,  // cell
                            _2,  // scratch
                            _3), // 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

https://www.dealii.org/8.5.1/doxygen/deal.II/namespaceWorkStream.html

NOTE: If your data objects are large, or their constructors are expensive, it is helpful to keep in mind that queue_length copies of the ScratchData object and queue_length*chunk_size copies of the CopyData object are generated.

Parallel computing model: MPI



General Considerations

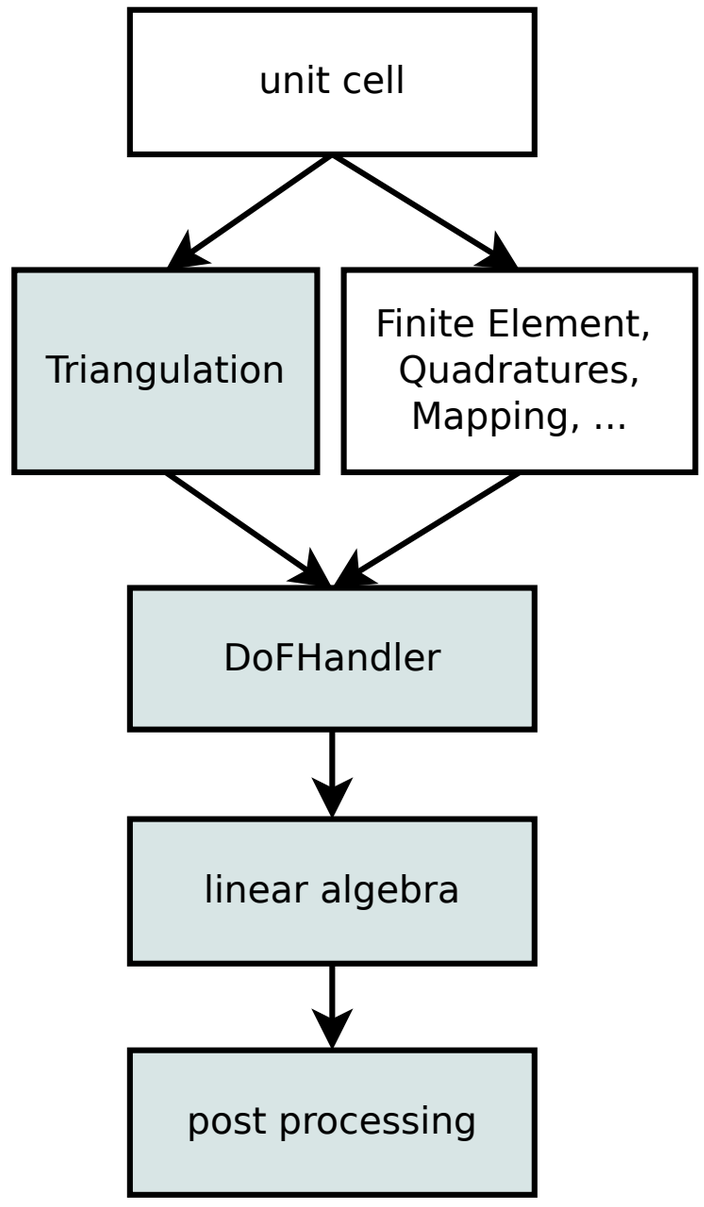
- Goal: get the solution faster!
- If FEM with <500.000 dofs, and 2d, use direct solver!

- 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:

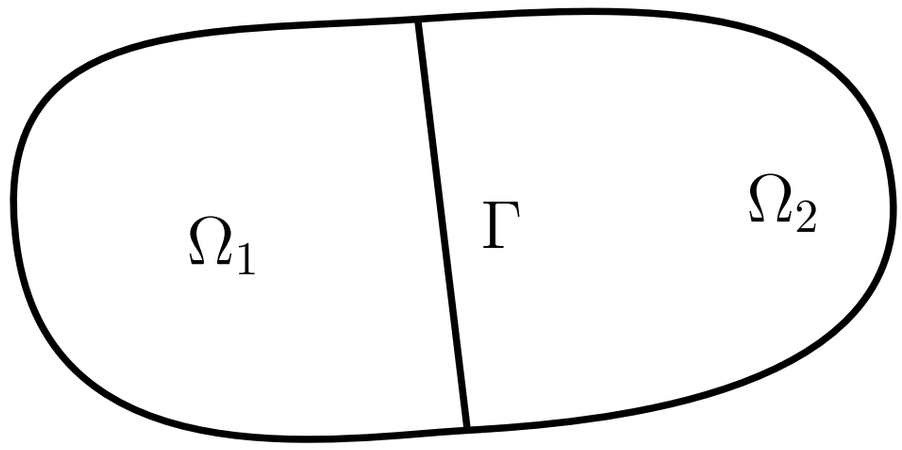
1. Triangulation (mesh with associated data)
   — hard: distributed storage, new algorithms
2. DoFHandler (manages degrees of freedom)
   — hard: find global numbering of DoFs
3. Linear Algebra (matrices, vectors, solvers)
   — use existing library
4. Postprocessing (error estimation, solution transfer,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



ma per seguir virtute e canoscenza
SISSA
ICTP

How to Parallelize?

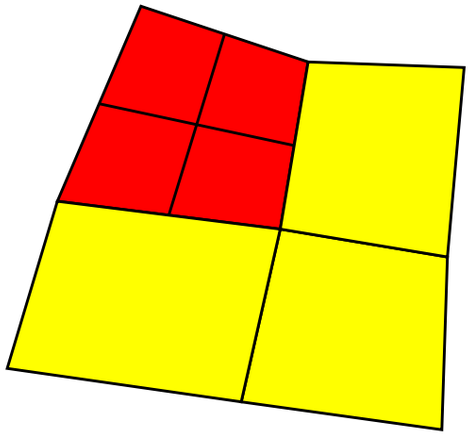
Option 1: Domain Decomposition



- 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!

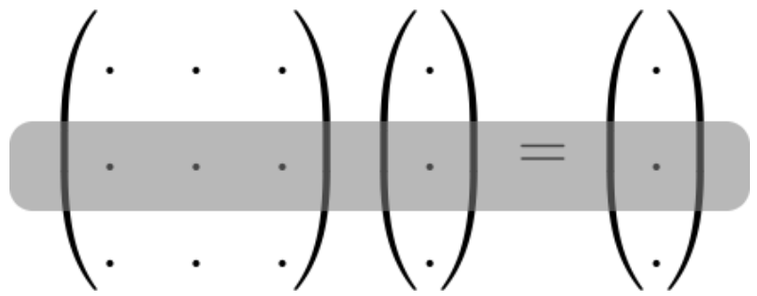
How to Parallelize?

Option 2: Algebraic Splitting



- Split up mesh between processors:

- Assemble logically global linear system (distributed storage):

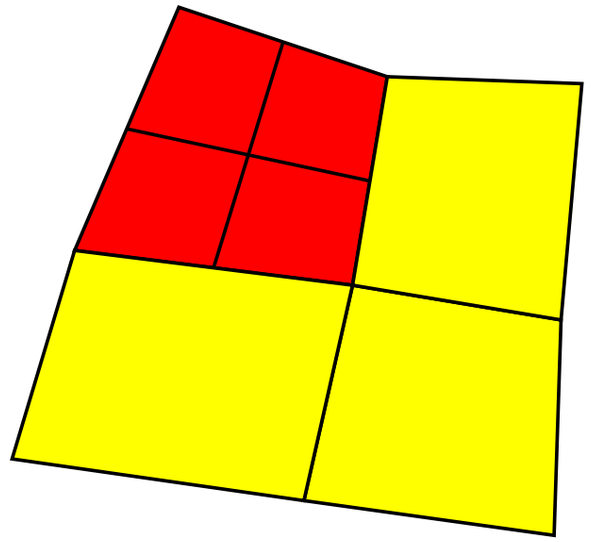


- Solve using iterative linear solvers in parallel
- Advantages:
  - Looks like serial program to the user
  - Linear scaling possible (with good preconditioner)

Partitioning

Optimal partitioning (coloring of cells):

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



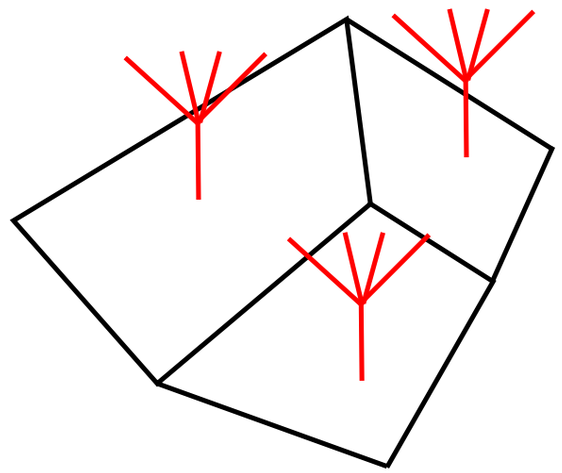
Optimal partitioning is an NP-hard graph partitioning problem.

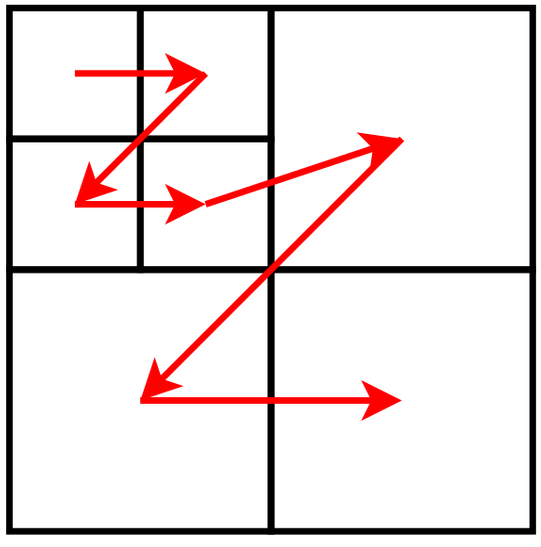
- Typically done: heuristics (existing tools: METIS)
- Problem: worse than linear runtime
- Large graphs: several minutes, memory restrictions

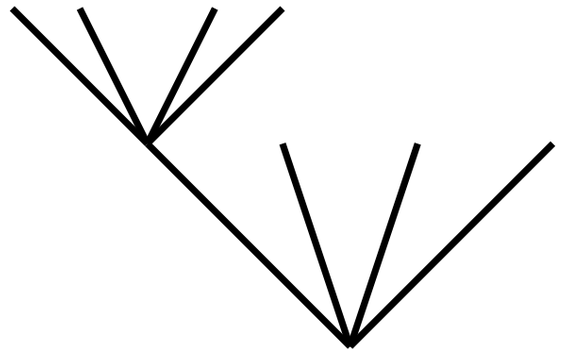
$\rightsquigarrow$ Alternative: avoid graph partitioning

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



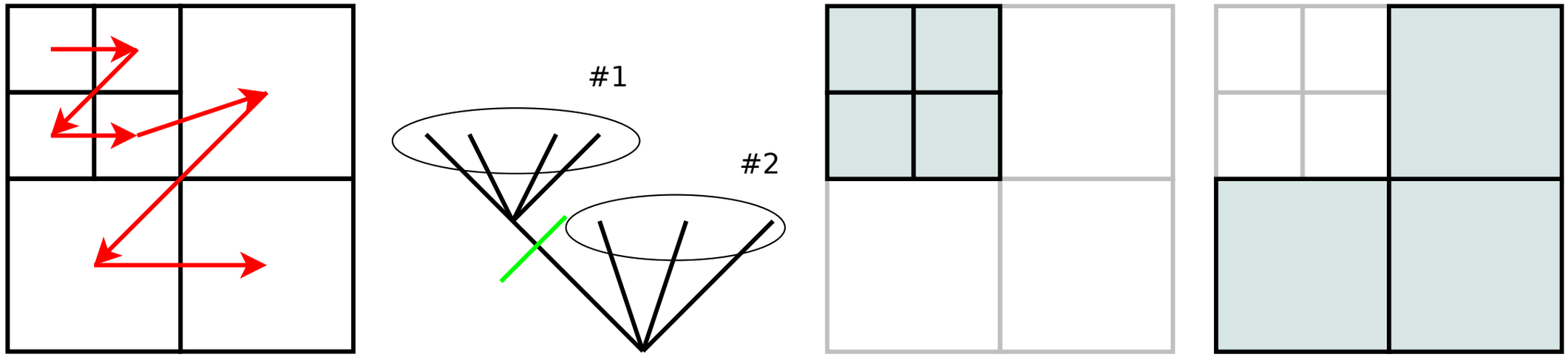




Burstedde, Wilcox, and Ghattas.
p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees.
SIAM J. Sci. Comput., 33 no. 3 (2011), pages 1103-1133.

Triangulation

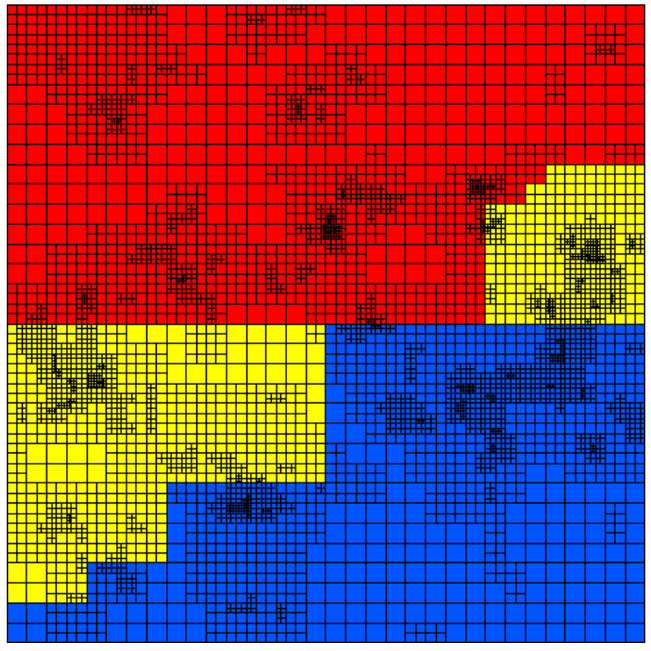
- Partitioning is cheap and simple:



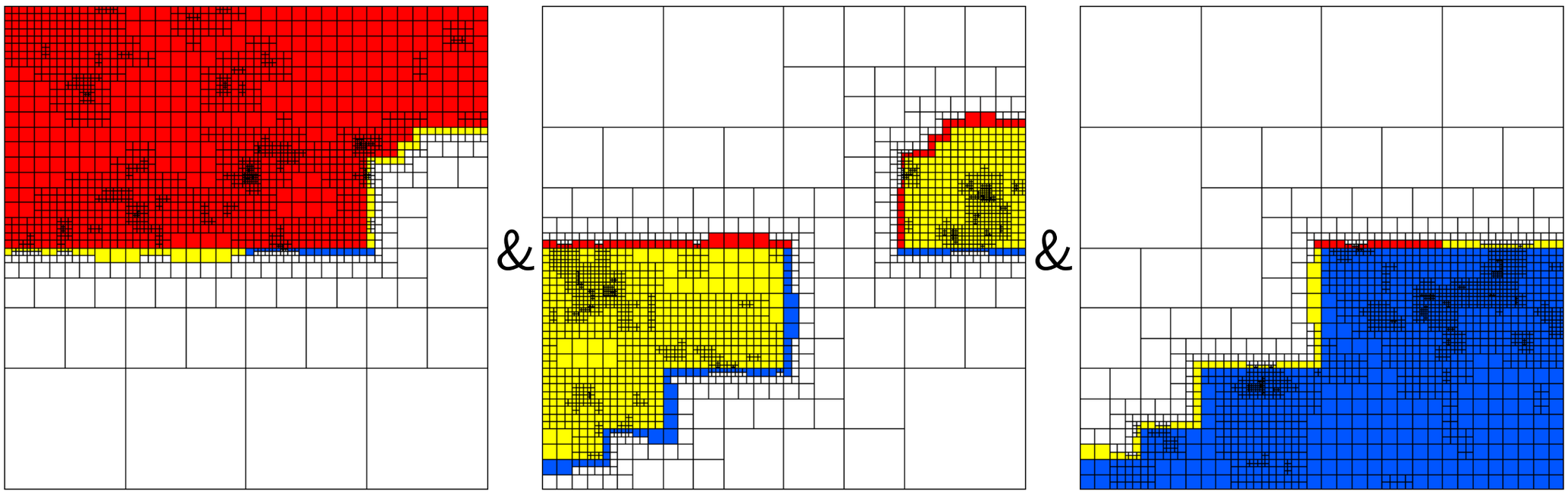
- Then: take *p4est* refinement information
- Recreate rich *deal.II* Triangulation only for local cells (stores coordinates, connectivity, faces, materials, . . .)
- How? recursive queries to *p4est*
- Also create ghost layer (one layer of cells around own ones)

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!

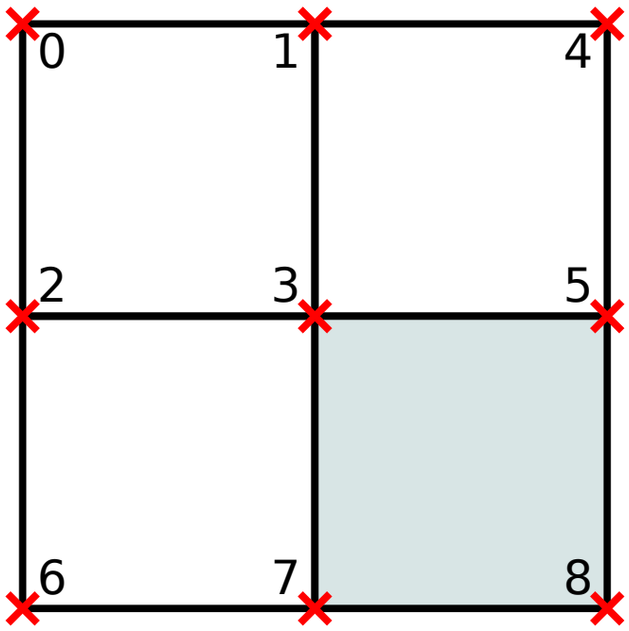
Master in High Performance Computing

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	?

ma per seguir virtute e canoscenza
SISSA
ICTP

Sketch...

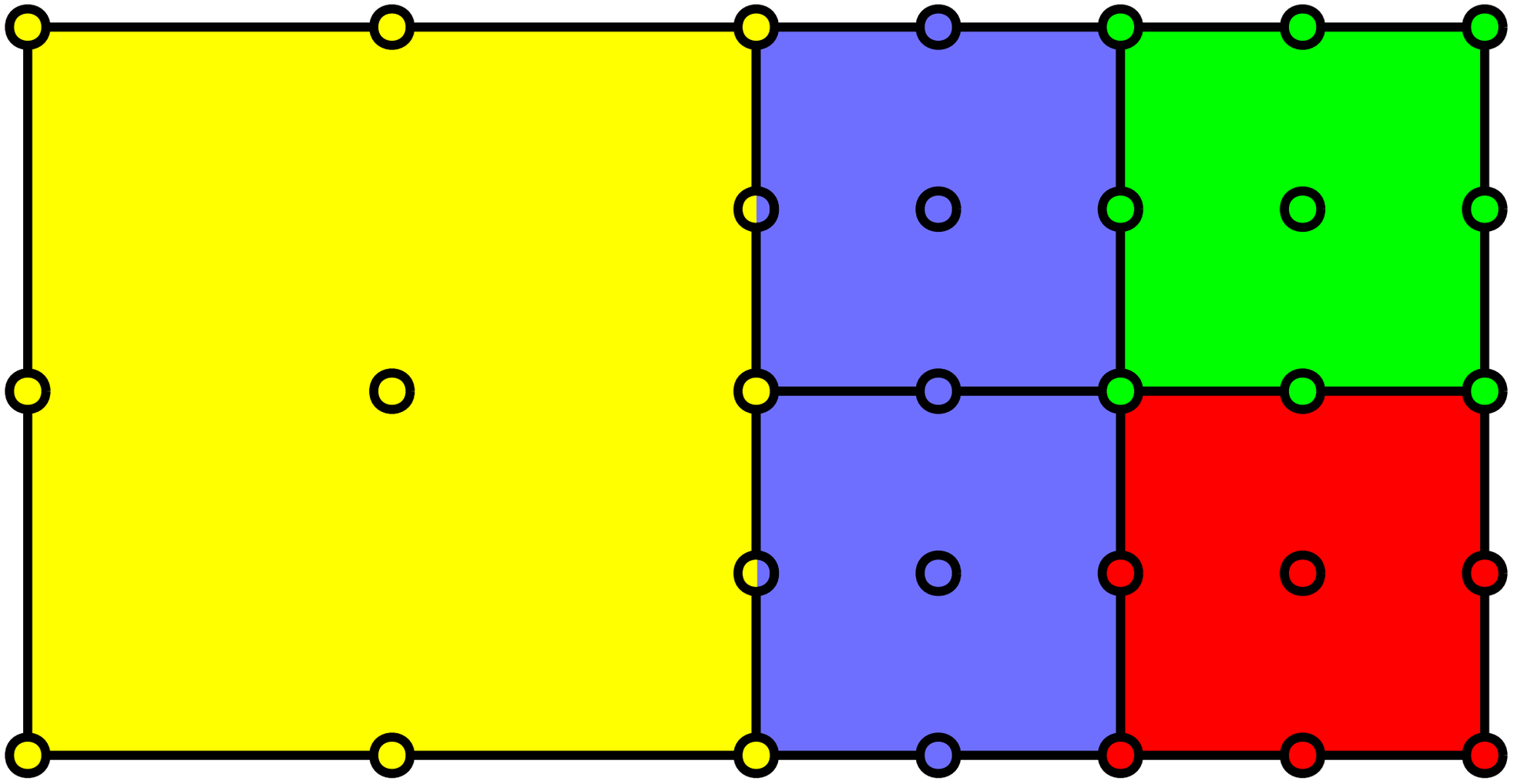


- Create global numbering for all DoFs
- Reason: identify shared ones
- Problem: no knowledge about the whole mesh

Sketch:

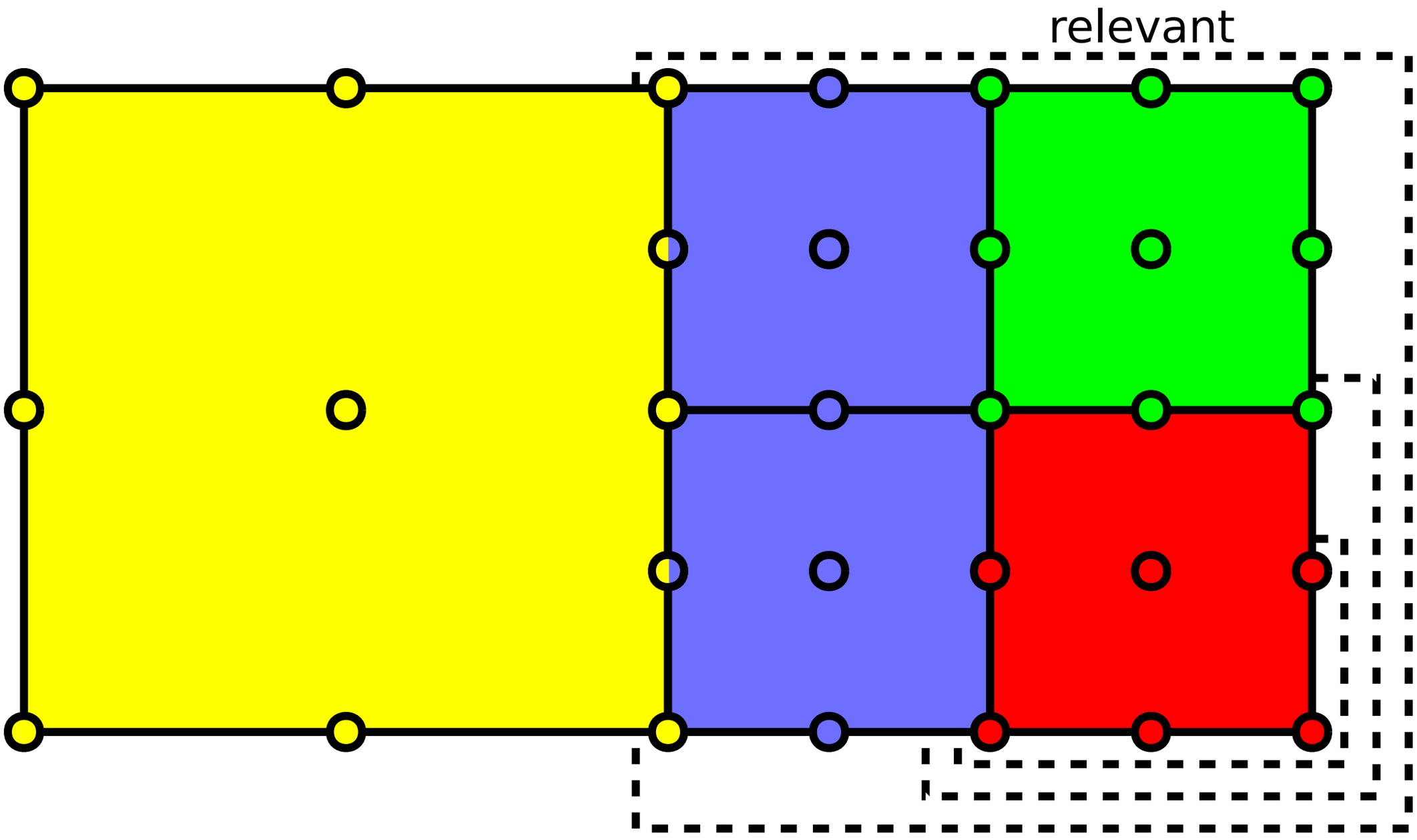
1. Decide on ownership of DoFs on interface (no communication!)
2. Enumerate locally (only own DoFs)
3. Shift indices to make them globally unique (only communicate local quantities)
4. Exchange indices to ghost neighbors

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

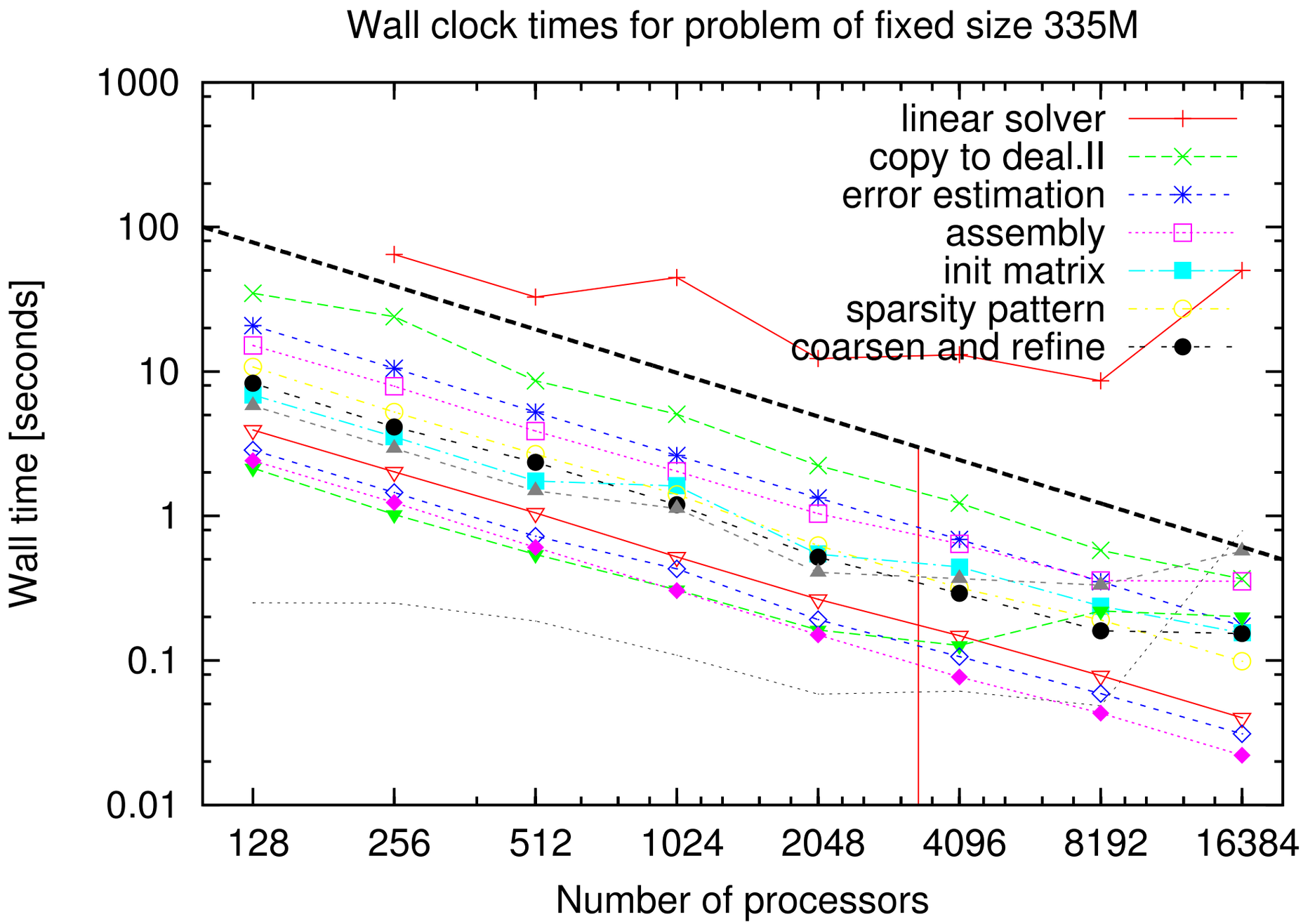
Trilinos VS PETSc

```
#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);

// ...
```

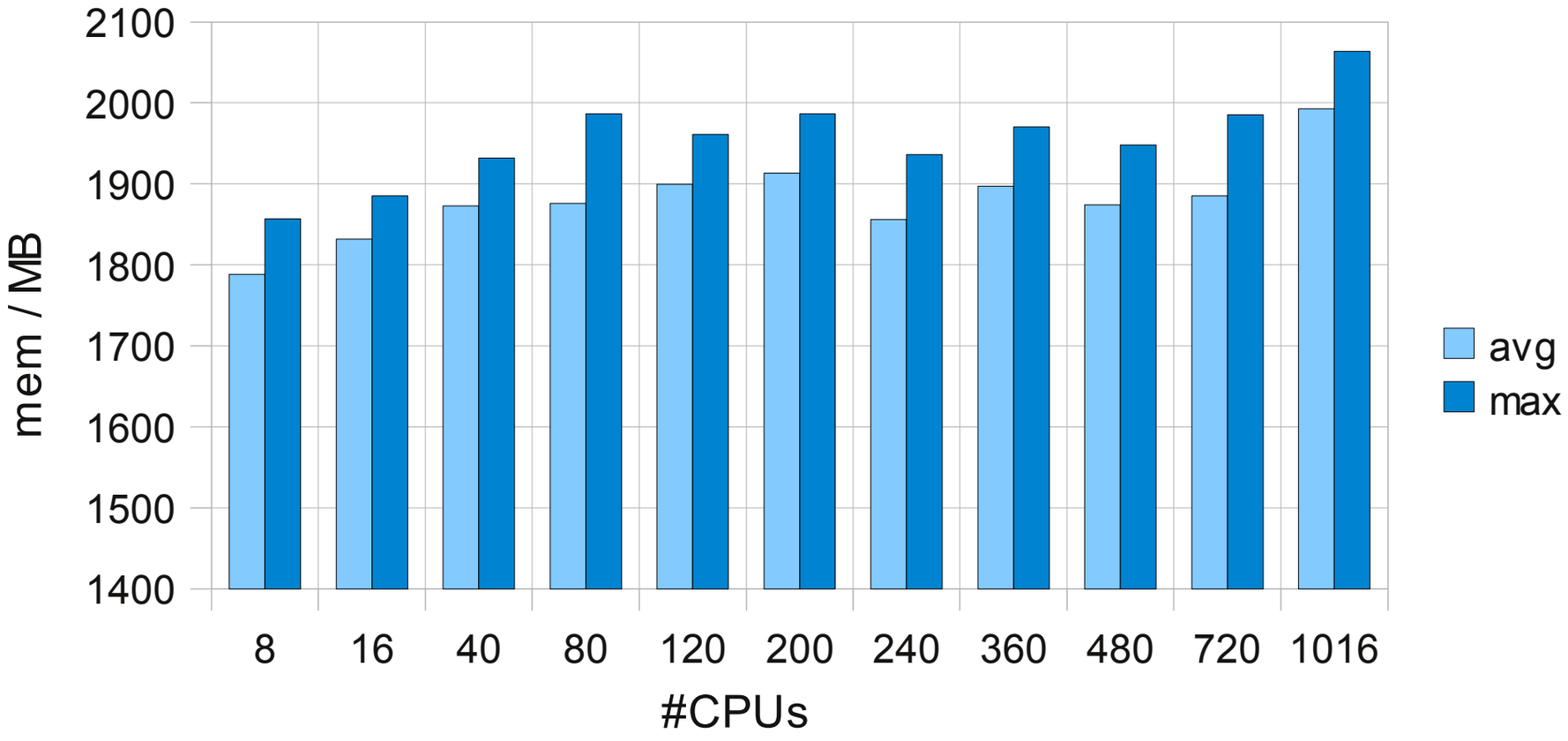
Solvers

- 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



Memory Consumption



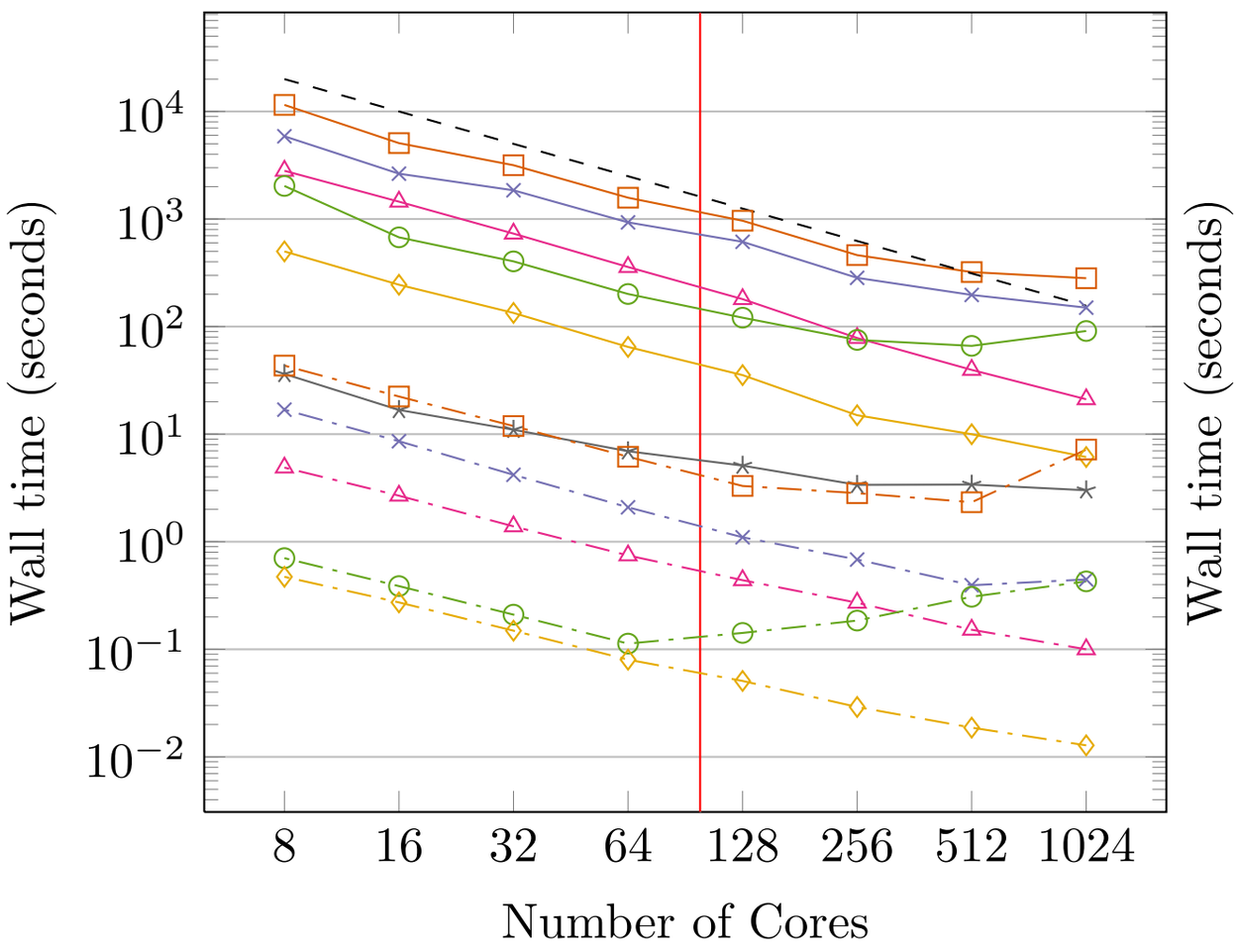
average and maximum memory consumption (VmPeak)
3D, weak scalability from 8 to 1000 processors with about 500.000 DoFs per processor (4 million up to 500 million total)

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

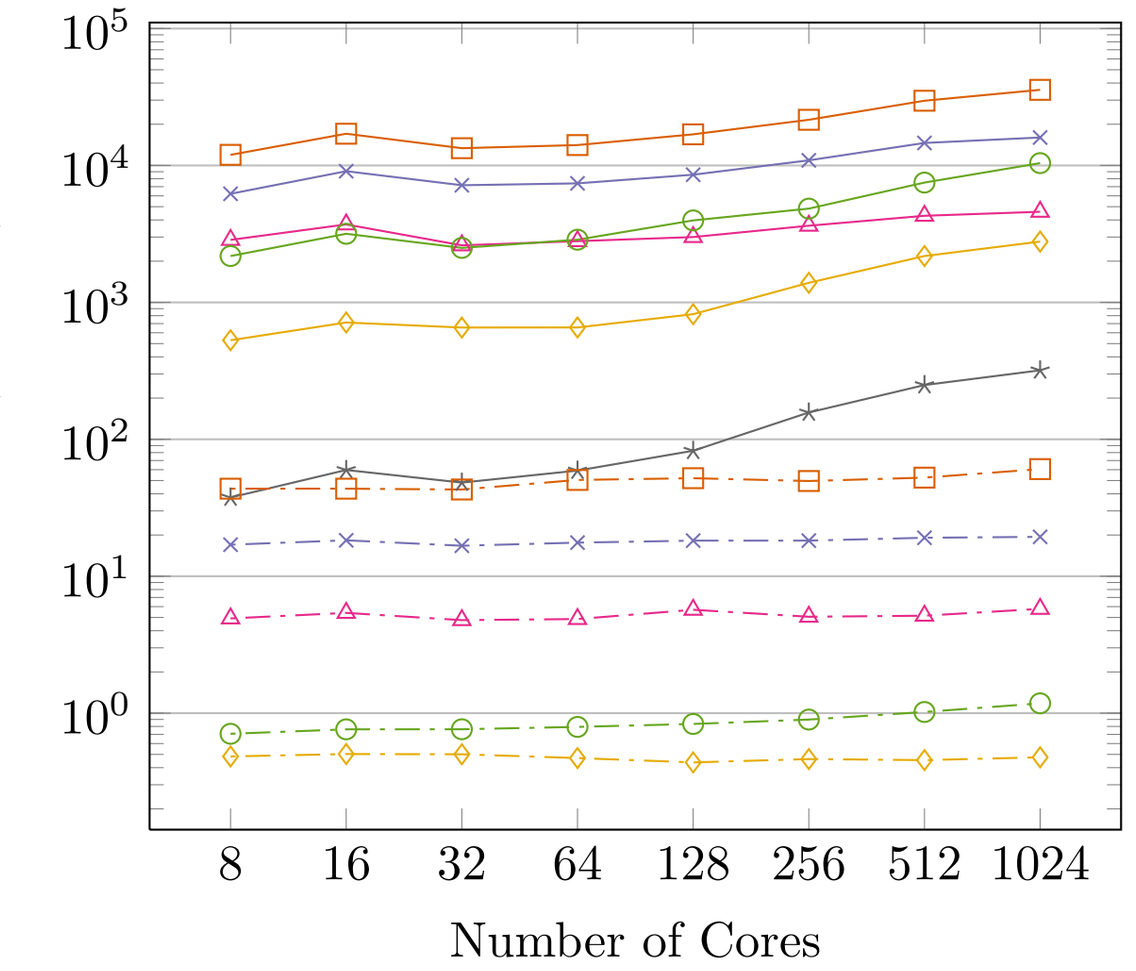
Master in High Performance Computing

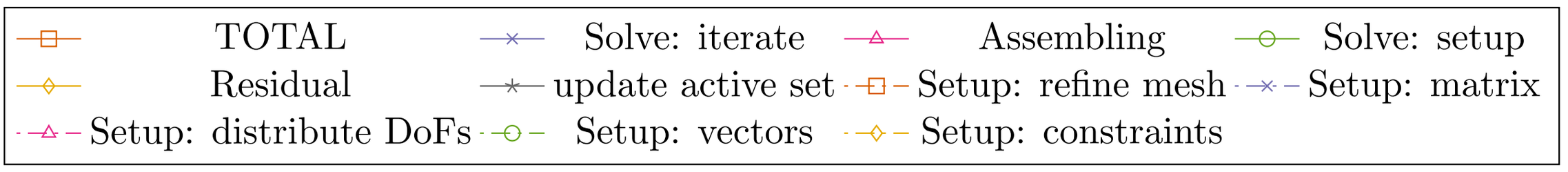
Step 40

Strong Scaling (9.9M DoFs)



Weak Scaling (1.2M DoFs/Core)





ma per seguir virtute e canoscenza
SISSA
ICTP