Using finite elements via the deal.II library

Wolfgang Bangerth, Texas A&M University

Lecture 1:

Course overview.

Why consider software libraries?

Course overview

The topic of this course:

Learn how to solve partial differential equations on computers! *

* Using the finite element method.

Course overview

The numerical solution of partial differential equations is an immensely practical field!

It requires us to know about:

- Partial differential equations
- Methods for discretizations, solvers, preconditioners
- Programming
- Adequate tools

Partial differential equations

Many of the big problems in scientific computing are described by partial differential equations (PDEs):

- Structural statics and dynamics
  – Bridges, roads, cars, …
- Fluid dynamics
  – Ships, pipe networks, …
- Aerodynamics
  – Cars, airplanes, rockets, …
- Plasma dynamics
  – Astrophysics, fusion energy
- But also in many other fields: Biology, finance, epidemiology, ...

On why to use existing software

There are times when we need to write computational software ourselves:

- When developing new computational methods
- When solving non-standard problems

In such cases, we could:

- Start from scratch, write everything ourselves
- Build something from existing components
- Adapt existing code written for similar applications

But: Option 1 could be difficult/time consuming/expensive!

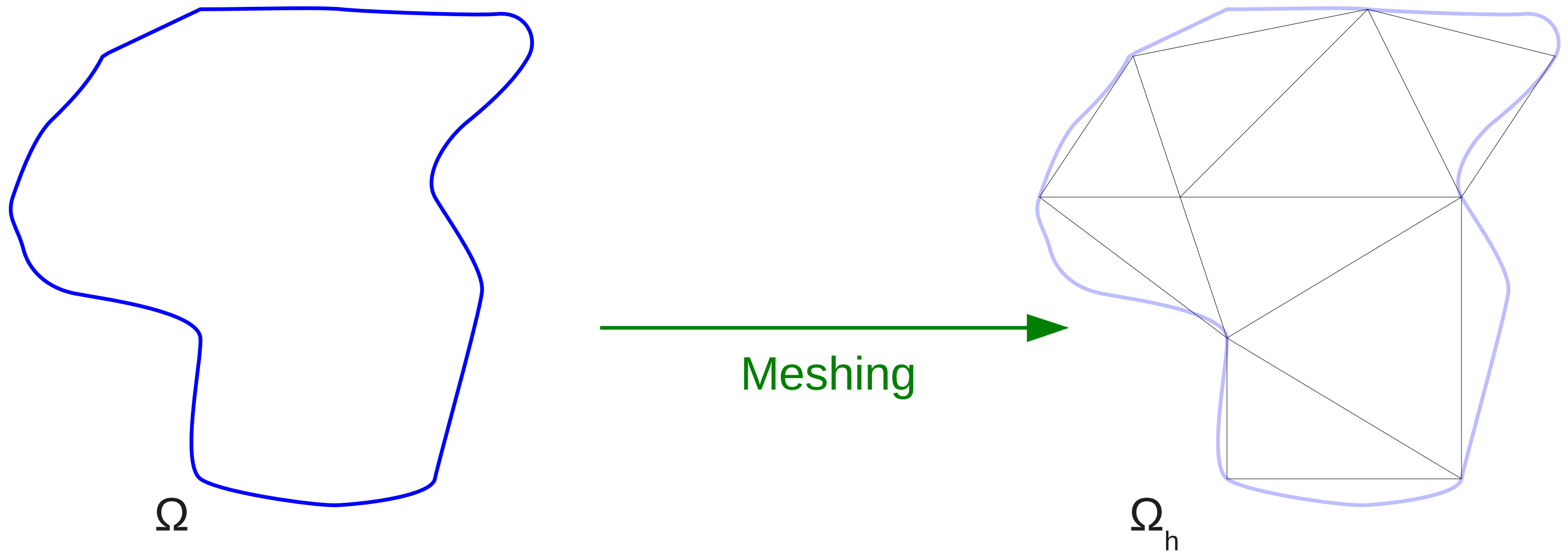
Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)



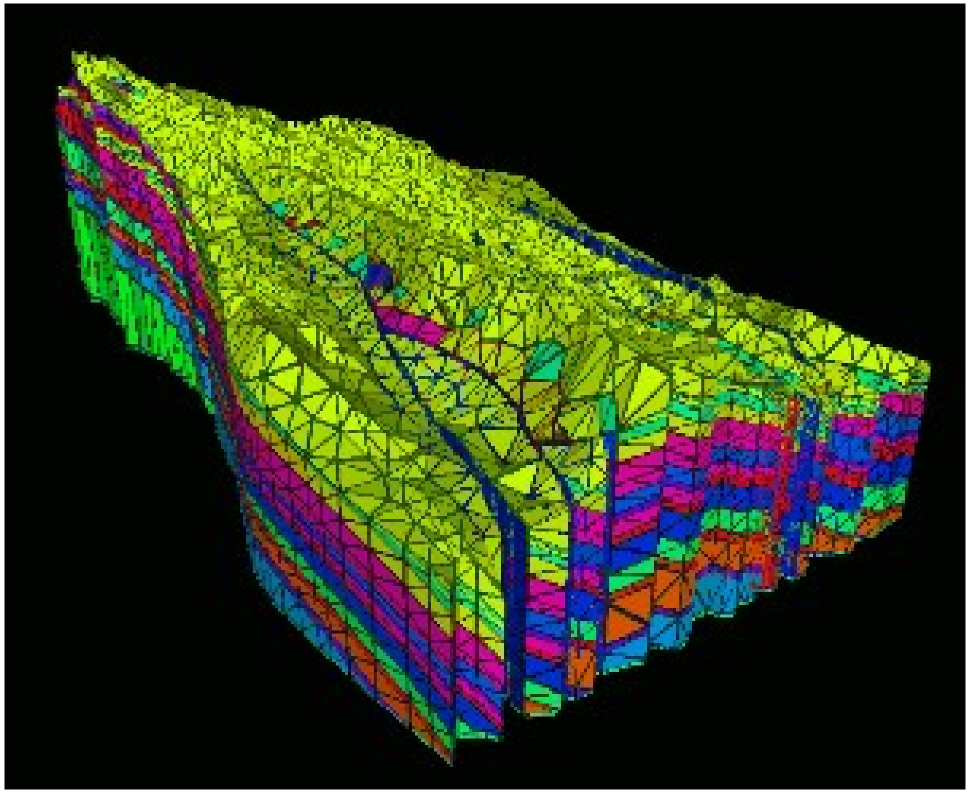
Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)



Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)
- Define balance relations on each cell
- Obtain and solve very large (non-)linear systems

Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)
- Define balance relations on each cell
- Obtain and solve very large (non-)linear systems

Problems:

- Every code has to implement these steps
- There is only so much time in a day
- There is only so much expertise anyone can have

Numerics for PDEs

Common features:

- Split the domain into small volumes (cells)
- Define balance relations on each cell
- Obtain and solve very large (non-)linear systems

Problems:

- Every code has to implement these steps
- There is only so much time in a day
- There is only so much expertise anyone can have

In addition:

- We don't just want a simple algorithm
- We want state-of-the-art methods for everything

Numerics for PDEs

Examples of what we would like to have:

- Adaptive meshes
- Realistic, complex geometries

- Quadratic or even higher order elements

- Multigrid solvers
- Scalability to 1000s of processors
- Efficient use of current hardware

- Graphical output suitable for high quality rendering

Q: How can we make all of this happen in a single code?

How we develop software

Q: How can we make all of this happen in a single code?

Not a question of feasibility but of how we develop software:

- Is every student developing their own software?
- Or are we re-using what others have done?

- Do we insist on implementing everything from scratch?
- Or do we build on existing libraries?

How we develop software

Q: How can we make all of this happen in a single code?

Not a question of feasibility but of how we develop software:

- Is every student developing their own software?
- Or are we re-using what others have done?

- Do we insist on implementing everything from scratch?
- Or do we build on existing libraries?

There has been a major shift on how we approach the second question in scientific computing over the past 10-15 years!

How we develop software

The secret to good scientific software is (re)using existing libraries!

Existing software

There is excellent software for almost every purpose!

Basic linear algebra (dense vectors, matrices):

- BLAS
- LAPACK

Parallel linear algebra (vectors, sparse matrices, solvers):

- PETSc
- Trilinos

Meshes, finite elements, etc:

- deal.II – the topic of this course
- …

Visualization, dealing with parameter files, ...

Existing software

Arguments against using other people's packages:

I would need to learn a new piece of software, how it works, its conventions. I would have to find my way around its documentation. Etc.

I think I'll be faster writing the code I want myself!

Existing software

Arguments against using other people's packages:

> I would need to learn a new piece of software, how it works, its conventions. I would have to find my way around its documentation. Etc.
>
> I think I'll be faster writing the code I want myself!

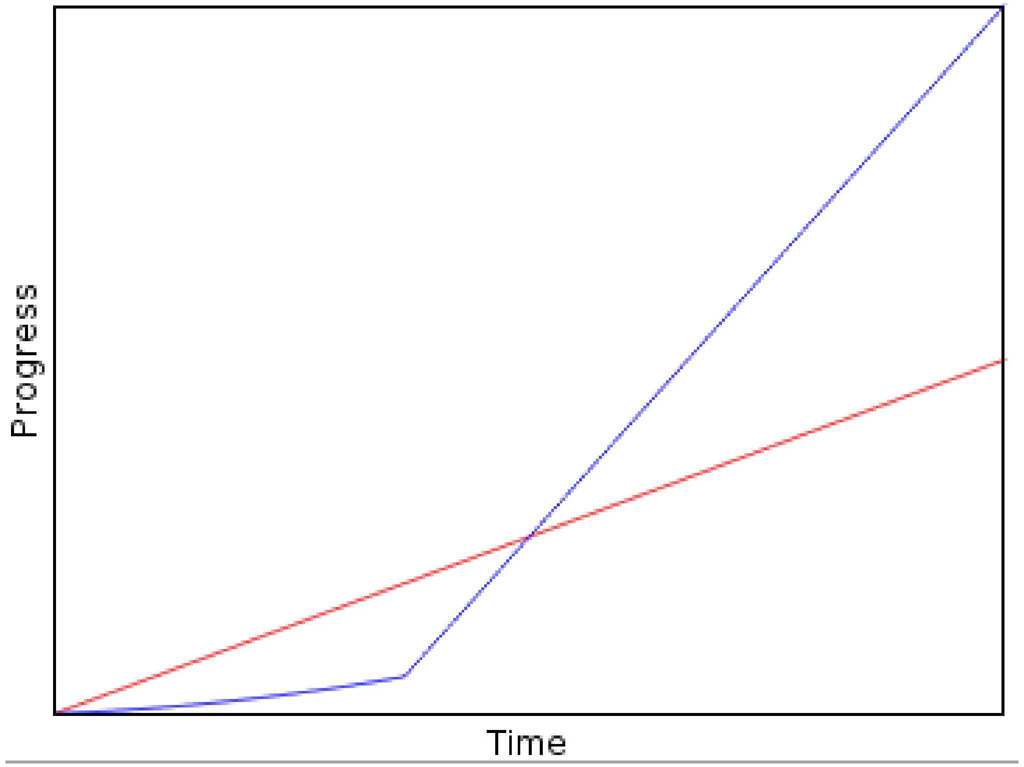
Answers:

- The first part is true.
- The second is not!

- You get to use a lot of functionality you could never in a lifetime implement yourself.
- Think of how we use Matlab today!

Existing software

Progress over time:

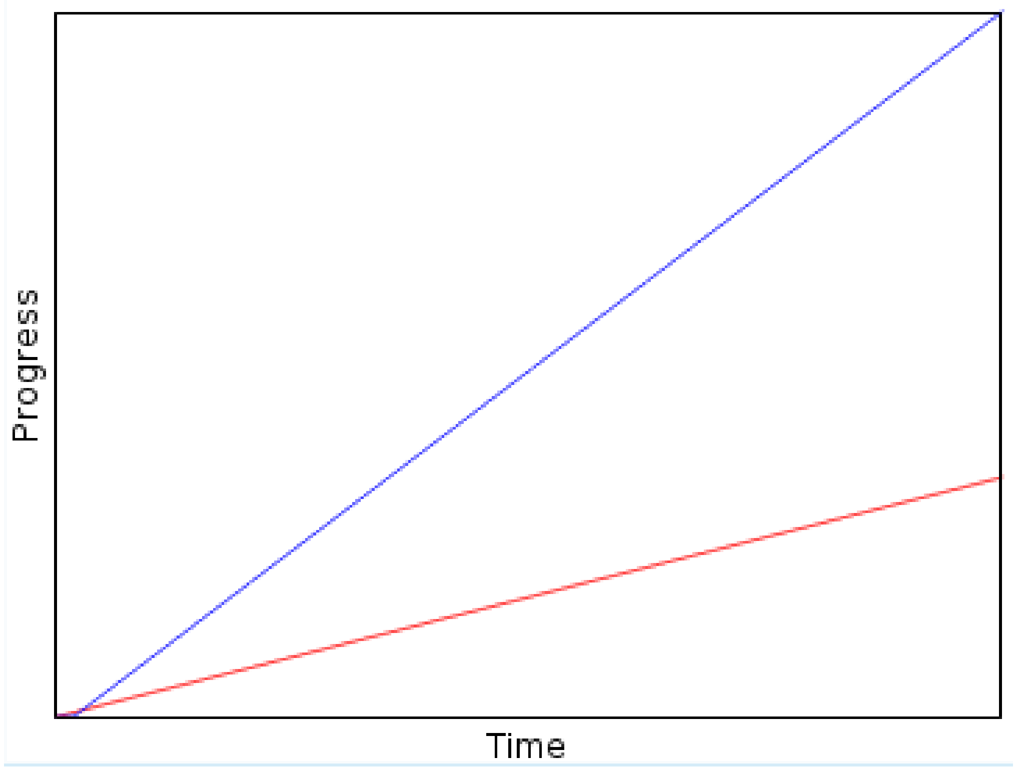


Red: Do it yourself. Blue: Use existing software.

Question: Where is the cross-over point?

Existing software

Progress over time, the real picture:



Red: Do it yourself.   Blue: Use existing software.

Answer: Cross-over is after 2–4 weeks! A PhD takes 3–4 years.

Existing software

Experience:

Students developing numerical methods can realistically expect to have a code at the end of a PhD time that:

- Works in 2d and 3d
- On complex geometries
- Uses higher order finite element methods
- Uses multigrid solvers or preconditioners
- Solves a nonlinear, time dependent problem

Doing this from scratch would take 10+ years.

Existing software

Arguments against using other people's packages:

How do I know that that software I'm supposed to use doesn't have bugs? How can I *trust* other people's software?

With my own software, at least I know that I don't have bugs!

Existing software

Arguments against using other people's packages:

How do I know that that software I'm supposed to use doesn't have bugs? How can I *trust* other people's software?

With my own software, at least I know that I don't have bugs!

Answer 1:

- You can't be serious to think that your own software has no bugs!

Existing software

Arguments against using other people's packages:

How do I know that that software I'm supposed to use doesn't have bugs? How can I *trust* other people's software?

With my own software, at least I know that I don't have bugs!

Answer 2:

- deal.II is developed by professionals with a lot of experience
- It has an extensive testsuite:

  We run 2,800+ tests after every single change!

Conclusions

- When having to implement software for a particular problem, re-use what others have done already
- There are many high-quality, open source software libraries for every purpose in scientific computing
- Use them:
  – You will be far more productive
  – You will be able to use state-of-the-art methods
  – You will have far fewer bugs in your code

If you are a graduate student:
Use them because you will be able to impress your adviser with quick results!

Lecture 2:

A real short overview of deal.II

deal.II

Deal.II is a finite element library. It provides:

- Meshes
- Finite elements, quadrature,
- Linear algebra
- Most everything you will ever need when writing a finite element code

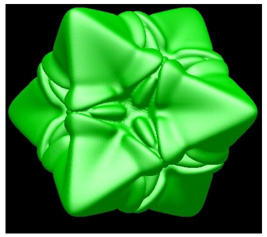
On the web at

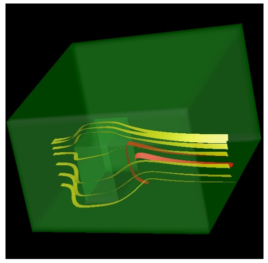
http://www.dealii.org/

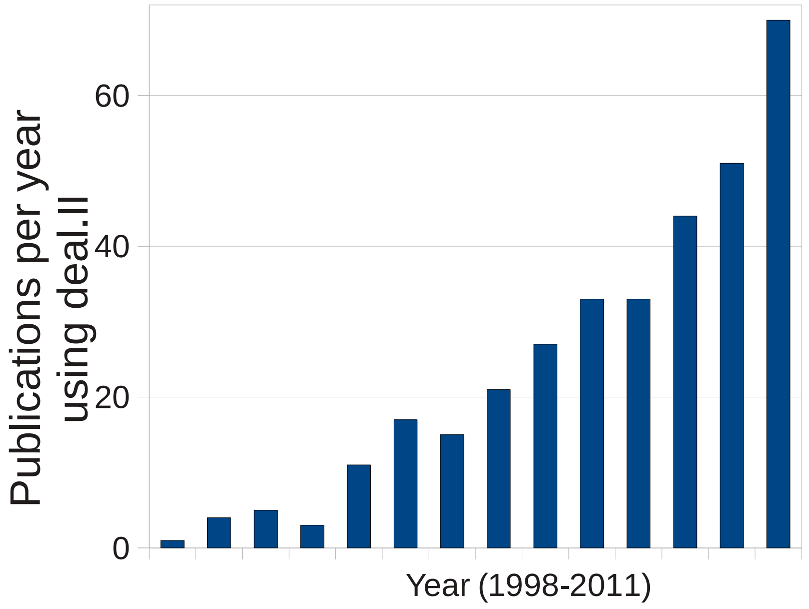
deal.II

deal.II is probably the largest FEM library:

- Presently ~600,000 lines of C++ code
- 10,000+ pages of documentation
- ~45 tutorial programs
- Fairly widely distributed:
  20,000+ downloads in 2012
- At least 65+ publications in 2012,
  400+ overall, that use it
- Used in teaching at a number
  of universities
- 2007 Wilkinson prize.



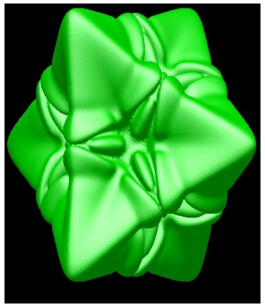


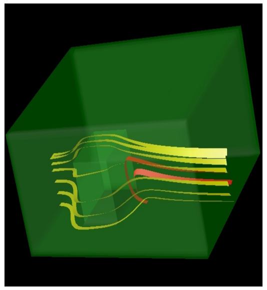


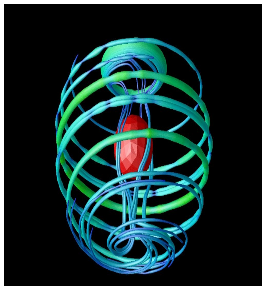
What's in deal.II

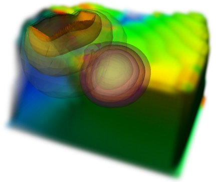
Meshes and elements:

- Supports adaptive meshes in 1d, 2d, and 3d
- Easy ways to adapt meshes: Standard refinement indicators already built in
- Many standard finite element types (continuous, discontinuous, mixed, Raviart-Thomas, ...)
- Low and high order elements
- Full support for multi-component problems





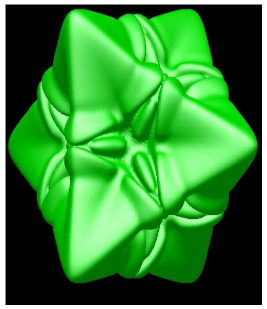


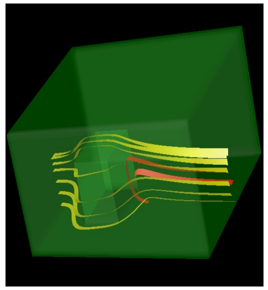


What's in deal.II

Linear algebra in deal.II:

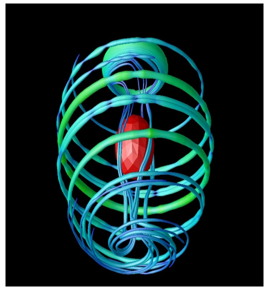
- Has its own sub-library for dense + sparse linear algebra
- Interfaces to PETSC, Trilinos, UMFPACK





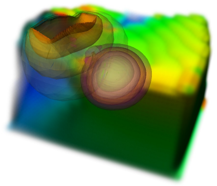
Pre- and postprocessing:

- Can read most mesh formats
- Can write almost any visualization file format



Parallelization:

- Uses threads and tasks on multicore machines
- Uses MPI, up to 10,000s of processors



What deal.II is used for

Apparently any PDE can be solved with deal.II.

In 2008–2010, papers were published that simulate:

- Biomedical imaging
- Heart muscle fibers

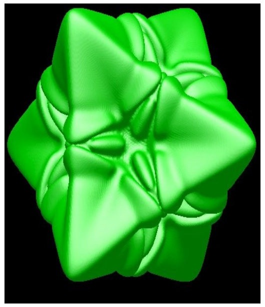
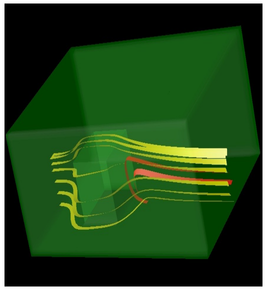
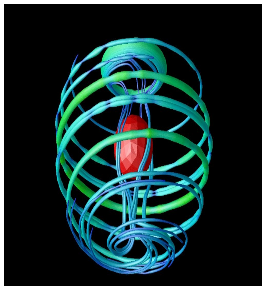
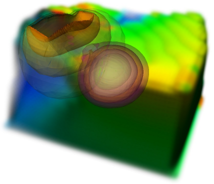
- Microfluidics
- Oil reservoir flow
- Fuel cells
- Aerodynamics

- Quantum mechanics
- Neutron transport

- Numerical methods research

- Fracture mechanics
- Damage models
- Sedimentation
- Biomechanics
- Root growth of plants
- Solidification of alloys
- Glacier mechanics

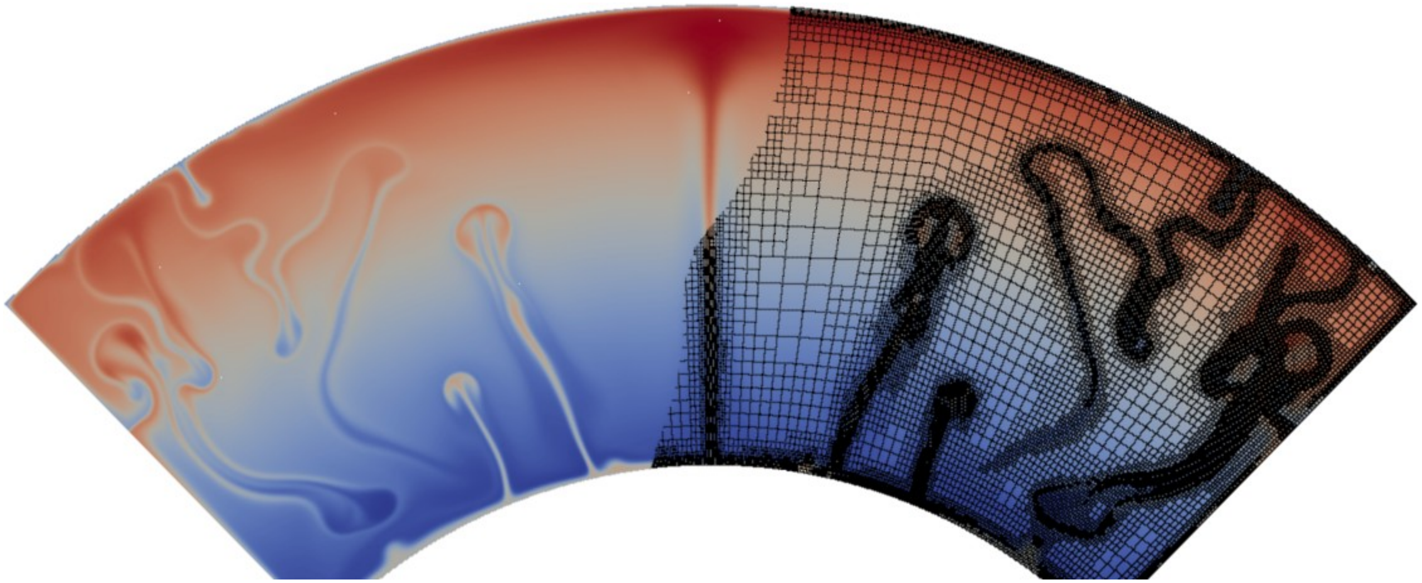
- Deterioration of statues due to air pollution

What deal.II is used for

Example: The mantle convection code ASPECT

http://aspect.dealii.org/



Methods:

- 2d, 3d, adaptive meshes, multigrid solvers
- Higher order finite elements
- Fully parallel

How deal.II is developed

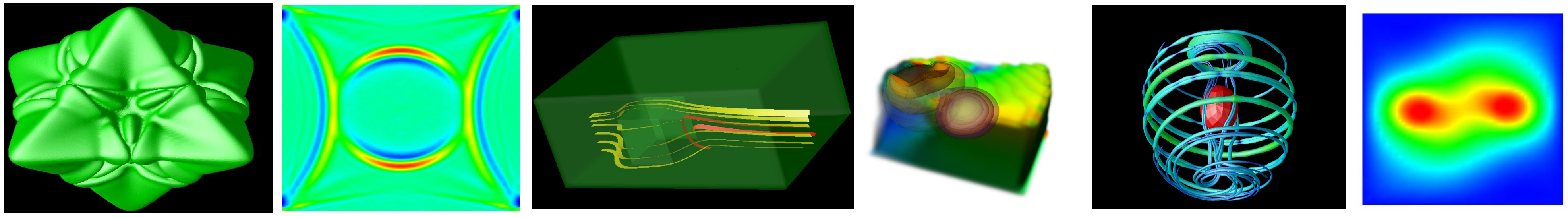
Development:

- 4–6 core developers (in the US, South Africa, Germany)
- ~10 occasional contributors (around the world)
- 100+ people have contributed over the past 10 years
- ~3000 lines of new code per month

deal.II is a typical open source project:

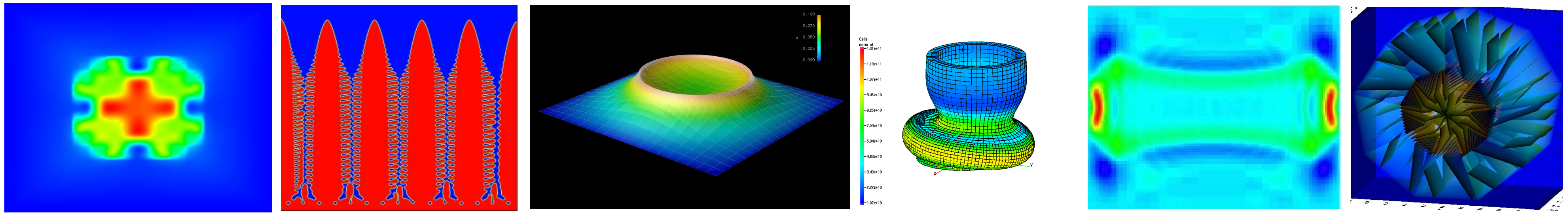
- People primarily develop what they need
- Open culture:
  – All development happens in the open
  – We (really) welcome everyone's contributions!

On the web



Visit the deal.II library:

http://www.dealii.org/



Conclusions

- **Mission:**
  To provide everything that is needed in finite element computations.

- **Development:**
  As an open source project

  As an inviting community to all who want to contribute

  As professional-grade software to users

Lecture 3:

The building blocks of a finite element code

Implementing the finite element method

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

We start with the strong form:

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

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:

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

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

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

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

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

If basis functions are linearly independent, this yields *N* equations for *N* coefficients.

Note: This is called the *Galerkin* method.

Implementing the finite element method

Practical question 1: How to define the basis functions?

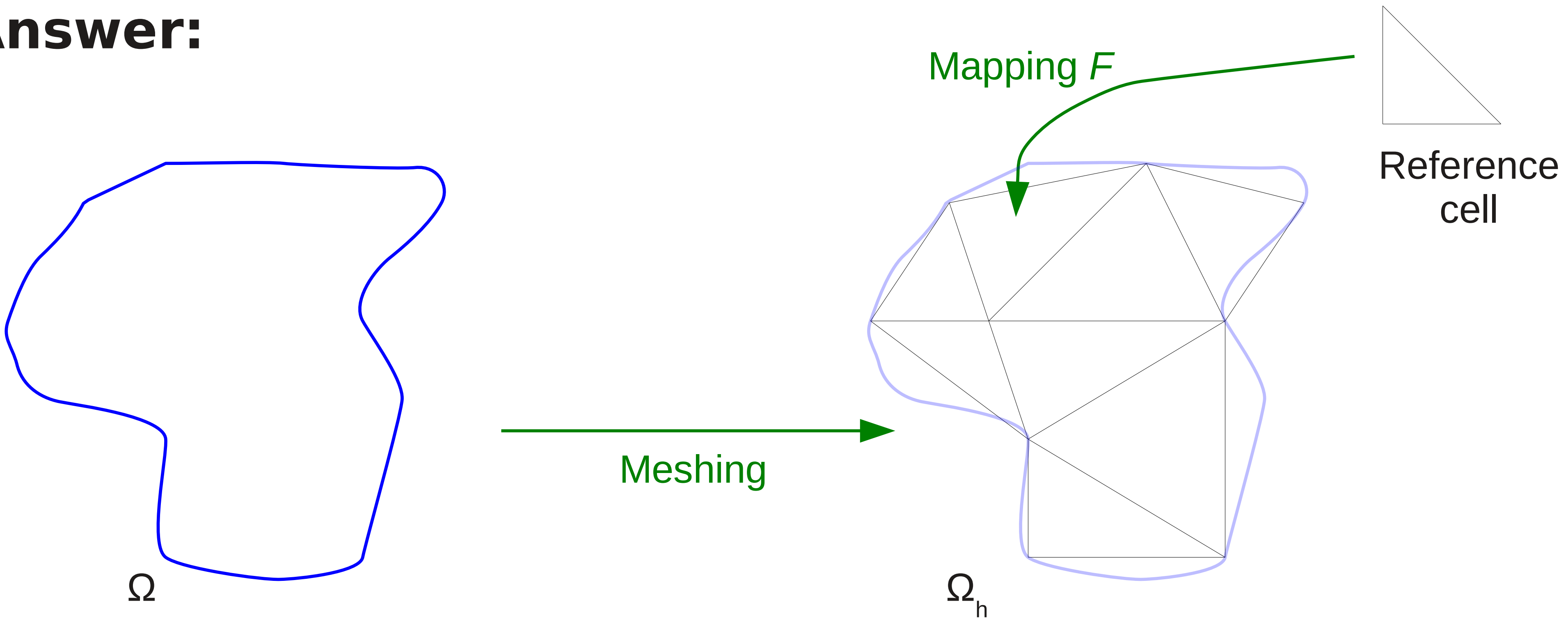
Answer: In the finite element method, this is done using the following concepts:

- Subdivision of the domain into a mesh
- Each cell of the mesh is mapped from 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?

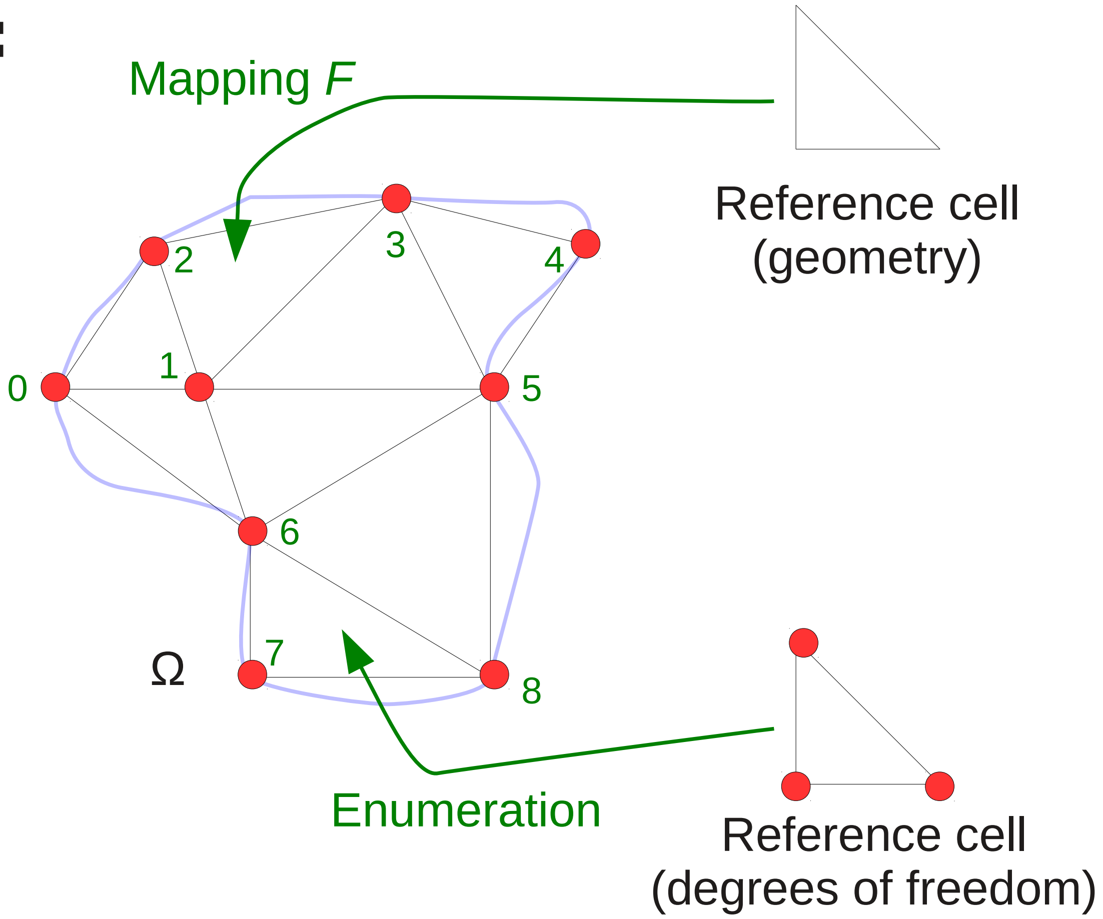
Answer:



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 mapped from 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 $u_h = \sum_{j=1}^{N} U_j \phi_j(x)$, we can expand the bilinear form

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

to obtain:

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

This is a linear system

$$AU = F$$

with

$$A_{ij} = (\nabla \phi_i, \nabla \phi_j) \qquad F_i = (\phi_i, f)$$

Implementing the finite element method

Practical question 2: How to compute

$$A_{ij} = (\nabla\phi_i, \nabla\phi_j) \qquad\qquad F_i = (\phi_i, f)$$

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

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

...and quadrature:

$$A_{ij} \approx \sum_K \sum_{q=1}^{Q} J_K^{-1}(\hat x_q)\hat\nabla\hat\phi_i(\hat x_q) \cdot J_K^{-1}(\hat x_q)\hat\nabla\hat\phi_j(\hat x_q)\ \underbrace{|\det J(\hat x_q)|\ w_q}_{=:\mathrm{JxW}}$$

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

$$AU=F$$

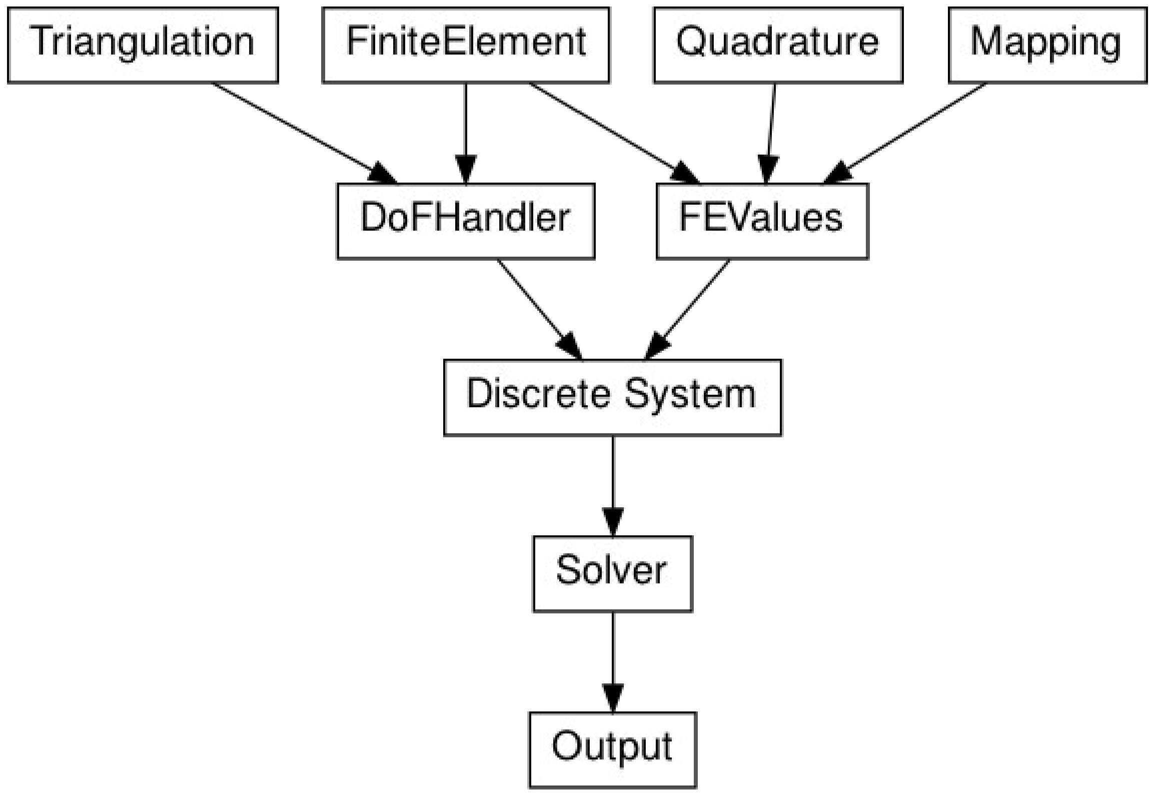
Answers: The goal is not to solve the linear system, but to do something with its solution:

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

These steps are often called *postprocessing the solution*.

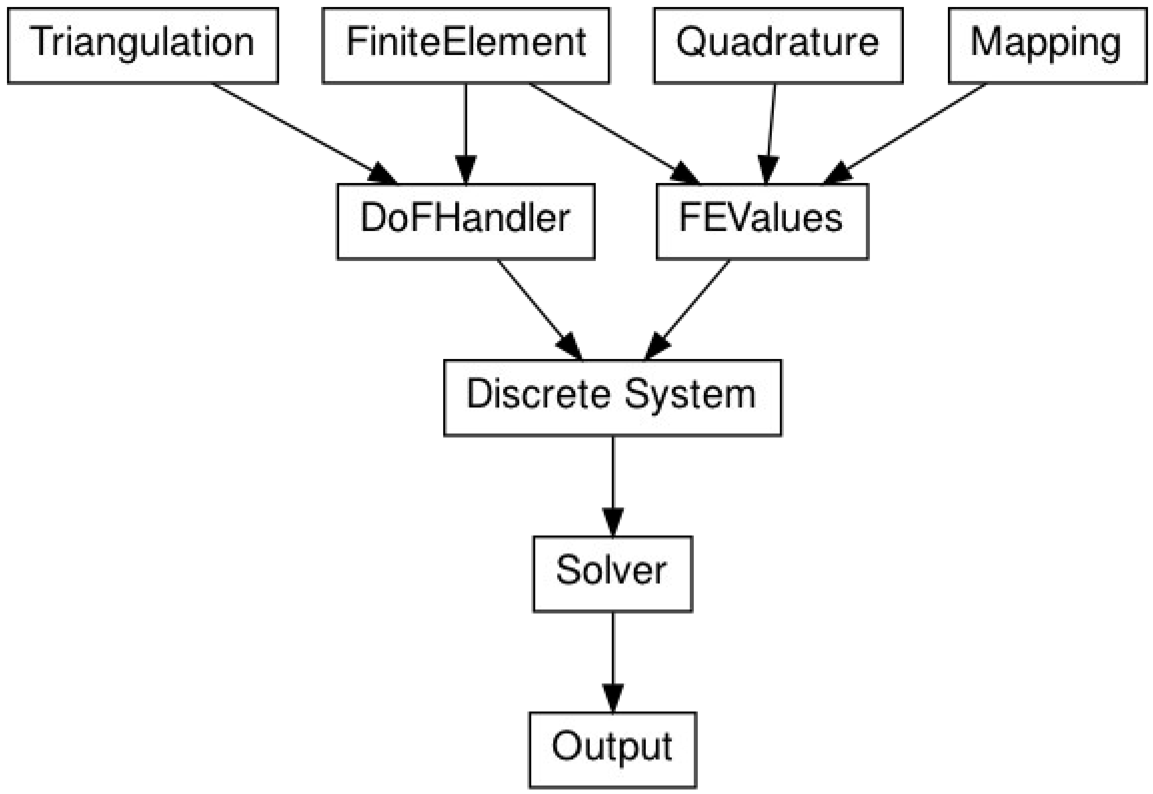
Implementing the finite element method

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

Each one of the components in this chart...



... can also be found in the *manual* at

http://www.dealii.org/7.2.0/index.html

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.

Lecture 4:

A first example

–

The *step-1* tutorial program: Triangulations

step-1

Step-1 shows:

- The *Triangulation* class
- How to think of a triangulation: as a collection of cells
- How to query cells for information, and what to do with them
- How to output a mesh, and a way to visualize it.

step-1

Tutorial programs have the following structure:

- Introduction:
  - lays out the problem to be solved
  - discusses the numerical method
  - introduces basics of the implementation
- Thoroughly documented code, processed for better readability
- Results section, often with suggestions for further extensions
- Copy of the code without the comments

All programs use similar structure and naming convention.

step-1

Read through the commented program at
http://www.dealii.org/7.1.0/doxygen/deal.II/step_1.html

Notes when reading:

- Read the introduction!
- If you want to *understand the entire code*, read from the top
- If you just want to *follow the flow* of the program, read from the bottom!
- Think about modifying the code as you read.

step-1

After reading, play with the program:

```
cd examples/step-1
cmake -DDEAL_II_DIR=/path/to/deal.II .
make run
```

This will run the program and generate output files:

```
ls -l
okular grid-2.eps
```

Next step: Play by following the suggestions in the results section. This is the best way to learn!

Lecture 5:

A second example:

The *step-2* tutorial program
–
Degrees of freedom (DoFs)

step-2

Step-2 shows:

- How degrees of freedom are defined with finite elements
- The *DoFHandler* class
- How DoFs are connected by bilinear forms
- Sparsity patterns of matrices
- How to visualize a sparsity pattern

step-2

Sparsity of system matrices:

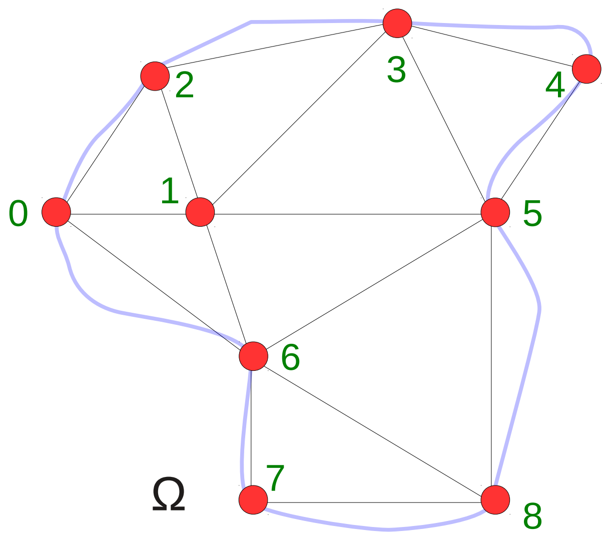
- For PDEs, finite element matrices are *always sparse*
- Result of
  – *local* definition of shape functions
  – *locality* of the differential operator

Sparsity is not a coincidence. It is a design choice of the finite element method.

Sparsity can not be overestimated as a factor in the success of the FEM!

step-2

Example: Consider this mesh and bilinear form:



$$A_{ij} = (\nabla\phi_i, \nabla\phi_j) = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j \, dx$$

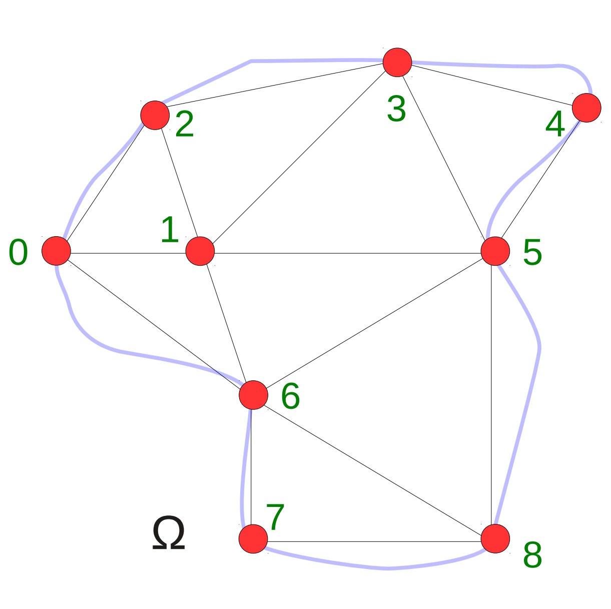
Note: In general we have that

- $A_{00} \neq 0, A_{01} \neq 0, A_{02} \neq 0, A_{06} \neq 0$
- $A_{03} = A_{04} = A_{05} = A_{07} = A_{08} = 0$

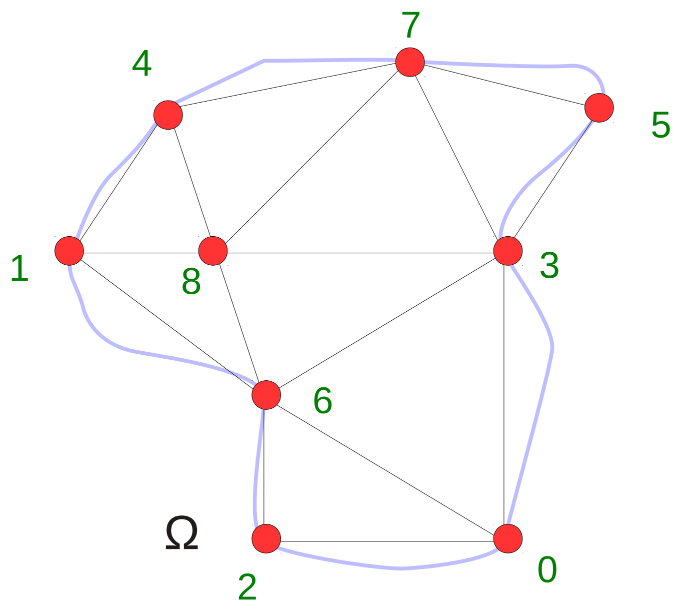
The bigger the mesh, the more zeros there are per row!

step-2

Renumbering: The order of enumerating degrees of freedom is arbitrary



vs.



Notes:

- Resulting matrices are just permutations of each other
- Both sparse, but some algorithms care

step-2

Read through the commented program at

http://www.dealii.org/7.1.0/doxygen/deal.II/step_2.html

Then play with the program:

```
cd examples/step-2
cmake -DDEAL_II_DIR=/a/b/c . ; make run
```

This will run the program and generate output files:

```
ls -l
```

Then run *gnuplot* as described in the documentation

```
gnuplot
```

Next step: Play by following the suggestions in the results section. This is the best way to learn!

Lecture 6:

A third example:

The *step-3* tutorial program – A first Laplace solver

step-3

Step-3 shows:

- How to set up a linear system
- How to assemble the linear system from the bilinear form:
  - The loop over all cells
  - The *FEValues* class
- Solving linear systems
- Visualizing the solution

step-3

Recall:

- For the Laplace equation, the bilinear form is written as a sum over all cells:

$$\begin{aligned} A_{ij} &= (\nabla \phi_i, \nabla \phi_j) \\ &= \sum_K \int_K \nabla \phi_i(x) \cdot \nabla \phi_j(x) \end{aligned}$$

step-3

Recall:

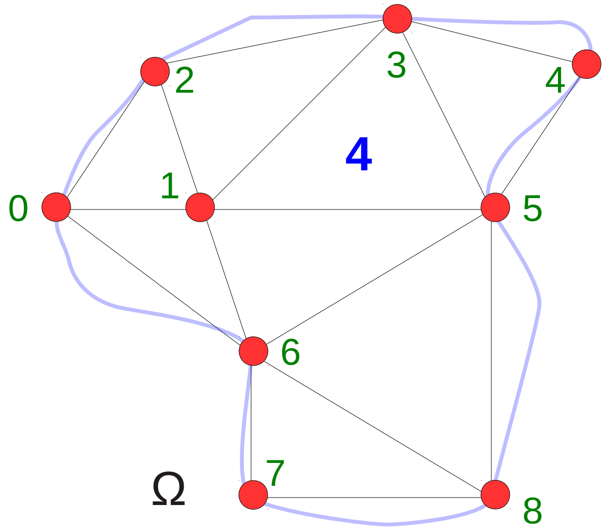
- For the Laplace equation, the bilinear form is written as a sum over all cells:

$$\begin{aligned} A_{ij} &= (\nabla \phi_i, \nabla \phi_j) \\ &= \sum_K \int_K \nabla \phi_i(x) \cdot \nabla \phi_j(x) \end{aligned}$$

- But on each cell, only few shape functions are nonzero!
- For Q_1, only $16=4^2$ matrix entries are nonzero per cell
- Only compute this (dense) sub-matrix, then “distribute” it to the global *A*
- Similar for the right hand side vector.

step-3

Example:



- On cell 4, only shape functions 1, 3, 5 are nonzero.
- We get a dense sub-matrix composed of rows and columns 1,3,5 of A.

step-3

Recall:

- We use quadrature

$$A_{ij}^K = \int_K \nabla \hat{\phi}_i(x) \cdot \nabla \hat{\phi}_j \, dx$$
$$\approx \sum_{q=1}^{Q} J_K^{-1}(\hat{x}_q) \hat{\nabla} \hat{\phi}_i(\hat{x}_q) \cdot J_K^{-1}(\hat{x}_q) \hat{\nabla} \hat{\phi}_j(\hat{x}_q) \underbrace{|\det J(\hat{x}_q)| \; w_q}_{=:\mathrm{JxW}}$$

- We really only have to evaluate shape functions, Jacobians, etc., at quadrature points – not as functions
- All evaluations happen on the reference cell

step-3

Read through the commented program at

http://www.dealii.org/7.1.0/doxygen/deal.II/step_3.html

Then play with the program:

```
cd examples/step-3
cmake -DDEAL_II_DIR=/a/b/c . ; make run
```

This will run the program and generate output files:

```
ls -l
```

Then run *visit* to visualize the output

```
visit
```

Next step: Play by following the suggestions in the results section. This is the best way to learn!