Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Implementing the finite element method

A 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

$$u_h = \sum_{i=1}^{N} U_i \varphi_i(x)$$

To determine the *N* coefficients we use the weak form and test with the *N* basis functions:

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

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

Implementing the finite element method

Practical question 1: How to define the basis functions?

Answer: In the finite element, this 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
- 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: In the finite element, this 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
- 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 expansion $u_h = \sum_{i=1}^{N} U_i \varphi_i(x)$, 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

$$AU = F$$

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...

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

...and quadrature:

$$A_{ij} \approx \sum_K \sum_{q=1}^{Q} J^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_i(\hat{x}_q) \cdot J^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_j(\hat{x}_q)(\hat{x}_q) \ |det\, J(\hat{x}_q)| \ w_q$$

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

$$AU = F$$

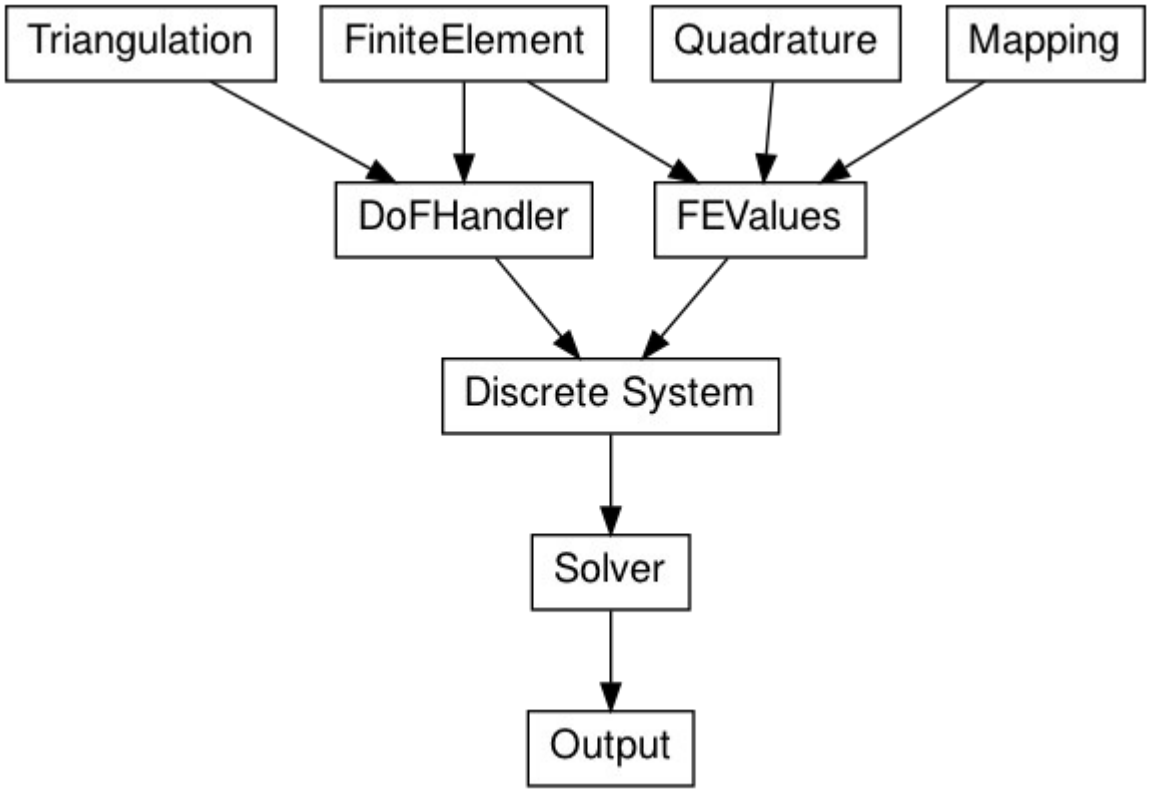
Answers: The goal is not to solve the linear system, but to do something with its solution:

- 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 collection of components that all appear (explicitly or implicitly) in finite element codes:



Implementation issues: Triangulations

Everyone's first triangulation implementation looks somewhat like this:

```
struct Vertex {    double coordinates[2];   };
struct Cell   {    int vertex_indices[3];   };

struct Triangulation {
  Vertex vertices[];
  Cell   cells[];
};
```

However, this is *not* a good design, for various reasons.

Implementation issues: Triangulations

Reason 1: You expose implementation details to the user of this class (lack of encapsulation):

```
struct Vertex {    double coordinates[2];   };
struct Cell   {    int vertex_indices[3];   };

struct Triangulation {
  Vertex vertices[];
  Cell   cells[];
};
```

Every piece of code that uses this triangulation makes use of the representation of data *in this particular form*. You can never again change it because you'd have to change things everywhere.

Implementation issues: Triangulations

Reason 2: This assumes that the mesh is static, i.e., that the number of vertices and cells never changes.

In reality, most modern codes today use adaptive mesh refinement in which the mesh changes all the time.

Implementation issues: Triangulations

Reason 3: In reality, we need to know much more about a triangulation:

```
struct Cell   {
    int   vertex_indices[3];
    int   neighbor_indices[3];
    int   material_index;
    int   subdomain_index;
    void* user_data;
    ...
};
```

The problem with this is that all this data is consecutive in memory, when we typically only want to access one field at a time for all cells. This leads to *cache misses*.

Implementation issues: Triangulations

A better approach: Identify which operations we need:

- Add and remove cells
- Iterate over all cells
- Ask cells for information

Note: Separate the interface from the data structure!

```
struct Cell   {
    int   vertex_index (int vertex) const;
    int   neighbor_index (int neighbor) const;
    int   material_index ();
    int   subdomain_index ();
    ...
};

struct Triangulation  {
    typedef … cell_iterator;    // acts like a Cell*
    cell_iterator begin () const;
    cell_iterator end () const;
};
```

Implementation issues: Triangulations

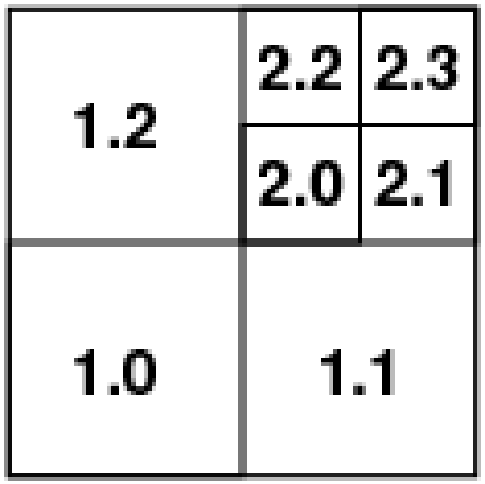
An example implementation:

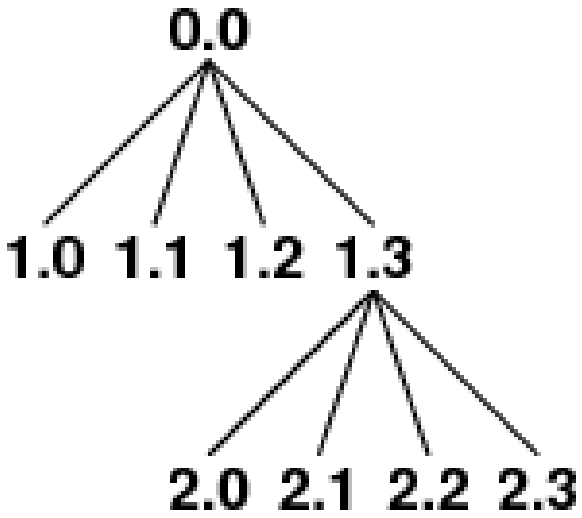
```
struct Triangulation  {
    cell_iterator begin () const { return cells.begin(); }
    ...
  private:
    std::list<Cell>     cells;
    std::vector<int>    cell_to_vertex_array;
    std::vector<Vertex> vertices;
};

struct Cell   {
    Vertex   vertex (int vertex) const {
      int vertex_index
          = tria->cell_to_vertex_array[index*3 + vertex];
      return tria->vertices[vertex_index];
    }
  private:
    Triangulation *tria;
    int index;
};
```

Implementation issues: Triangulations

How triangulations are really stored today:

An adaptively refined mesh starting from a single cell can be considered a *quad-tree*!



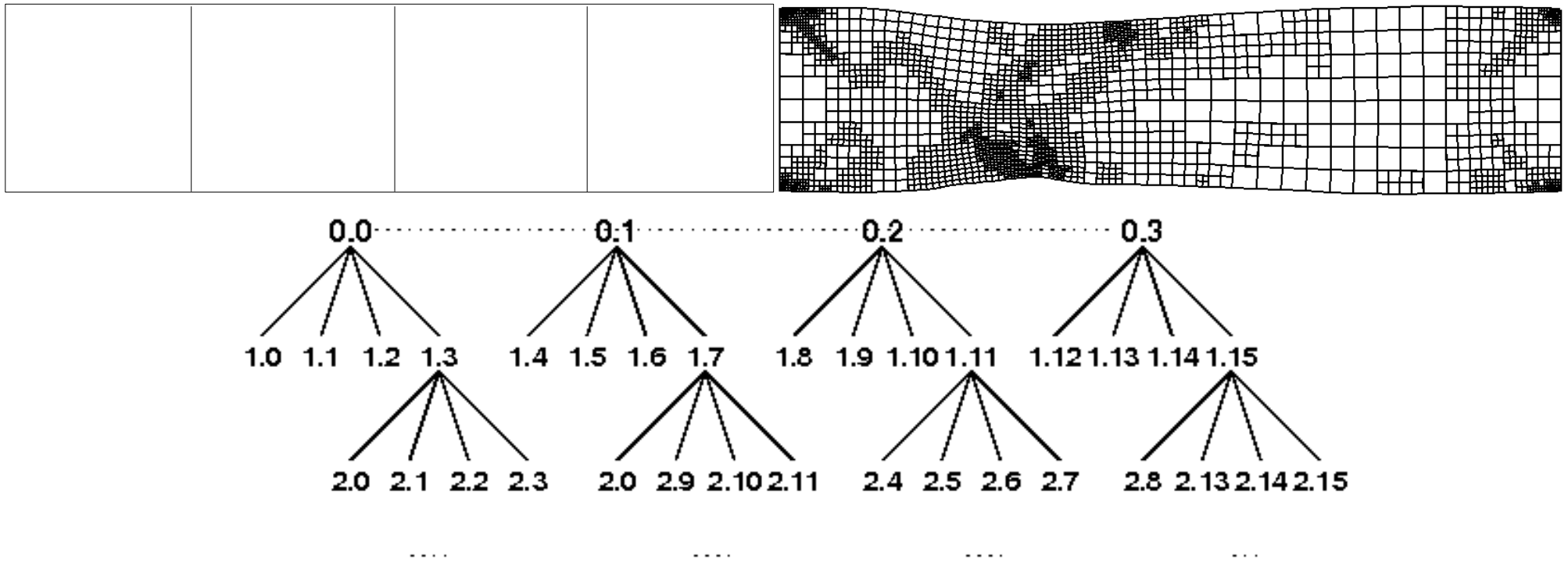


The 3d equivalent of this tree is an *octree*.

Implementation issues: Triangulations

How triangulations are really stored today:

An adaptively refined mesh starting from an unstructured coarse mesh is a *quad-forest*!



The 3d equivalent of this tree is an *oct-forest*.

Implementation issues: Finite elements

While most finite element codes have only one implementation of the Triangulation class, they typically have many different element types implemented:

```
class FiniteElement {          // an interface class
  int dofs_per_vertex () const = 0;
  int dofs_per_edge () const = 0;
  int dofs_per_triangle () const = 0;

  double shape_value (int i, Point p) const = 0;
  … …    shape_grad  (int i, Point p) const = 0;
};

class FE_Lagrange      : public FiniteElement {…};
class FE_RaviartThomas : public FiniteElement {…};
class FE_Nedelec       : public FiniteElement {…};
… …
```

Implementation issues: Quadrature/Mapping

The same is true for quadrature objects:

```
class Quadrature {          // an interface class
  Point  quadrature_point (int q) const = 0;
  Double quadrature_weight (int q) const = 0;
};

class Q_Gauss        : public Quadrature {…};
class Q_Trapezoidal : public Quadrature {…};
… …
```

Mapping classes are implemented similarly, providing linear, quadrature, … cartesian, mappings.

Implementation issues: FEValues objects

Remember that our integration procedure looked like this:

$$A_{ij} \approx \sum_K \sum_{q=1}^{Q} J^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_i(\hat{x}_q) \cdot J^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_j(\hat{x}_q)(\hat{x}_q) \; |det\, J(\hat{x}_q)| \; w_q$$

Note: This formula references

- Shape functions (the finite element)
- Jacobians (the mapping)
- Quadrature points and weights (the quadrature)

It turns out that in practice, one never references

- Shape functions without mappings
- Mappings with shape functions
- Shape functions and mappings without quadrature

FEValues is a way to present the application with an interface to exactly the things it needs (not 3 interfaces).

Implementation issues: FEValues objects

Remember that our integration procedure looked like this:

$$A_{ij} \approx \sum_K \sum_{q=1}^{Q} J^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_i(\hat{x}_q) \cdot J^{-1}(\hat{x}_q)\hat{\nabla}\hat{\varphi}_j(\hat{x}_q)(\hat{x}_q) \; |det\, J(\hat{x}_q)| \; w_q$$

Note:

- Some of these terms change from cell to cell

$$J^{-1}(\hat{x}_q), \; |det\, J(\hat{x}_q)|$$

- Some are always the same provided we use the same shape functions and quadrature points on each cell:

$$\nabla\hat{\varphi}_i(\hat{x}_q), \; w_q$$

- Furthermore, even in the computation of the variable components, some parts may always be the same.

Efficient codes should cache the stable components!

Implementation issues: FEValues objects

In deal.II, the **FEValues** class is such a cache:

- At the beginning of a loop over all cells, it computes the immutable components once (values, gradients on reference cell)
- Whenever we move from one cell to the next, it re-computes the variable parts (things that depend on the location of vertices)
- In fact, even for the latter class it analyzes whether the next cell is similar to the previous one to save computations. E.g.:
  - If the next cell is just a translation of the previous one, then the Jacobian matrix is the same.
  - If it is a translation + rotation, then the determinant of the Jacobian is the same

Implementation issues: Linear algebra

Appropriate data structures for vectors are obvious: Arrays.

For sparse matrices, one typically uses the *compressed sparse compressed* (CSR) format:

- Have one long integer array in which we store the column numbers of all nonzero entries in the matrix
- Have one equally long floating point array in which we store the values
- Have one array that indicates the beginning of each row

Implementation issues: Linear algebra

Compressed sparse compressed (CSR) example:

$$\begin{pmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{pmatrix}$$

With zero-based indexing:

- "Rowstart" array:

0, 2, 5, 8	10

- "Colnum" array:

0,1	0,1,2	1,2,3	2,3

- "Values" array:

2,-1	-1,2,-1	-1,2,-1	-1,2

Finding an array costs *O(log m)* where *m*=bandwidth.

Matrix-vector product costs *O(Nm).*

Sometimes one sorts the diagonal to the front of each row.

Implementation issues: Solving systems

Solvers and preconditioners:

For "simple" problems with up to 100,000 unknowns:

- Can use iterative solvers such as CG/GMRES/...
- Can use *sparse direct solvers* (such as UMFPACK, Matlab's \-operator)
- Sparse direct solvers often faster, always work

For problems with up to a few million unknowns:

- CG/GMRES with "simple preconditioners" (Jacobi, SSOR)

For "big" problems (several million to billions of unknowns):

- CG/GMRES
- We need a parallelizable preconditioner (AMG, block decompositions)

The bigger picture

Numerical analysis and finite element/difference/volume methods are only one piece in the scientific computing world.

The *goal* is always the simulation of real processes for *prediction and optimization*.

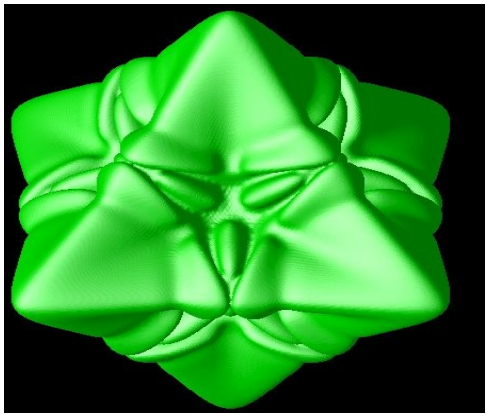
This also involves:

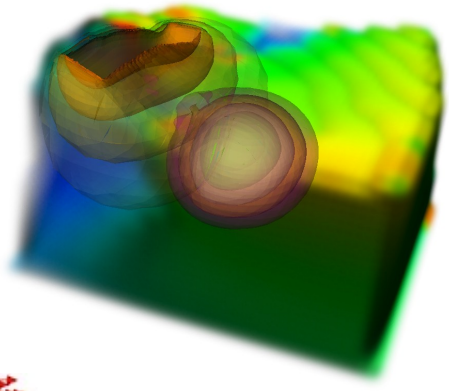
- Understanding the application
- Implementation of numerical methods
- Understanding the complexity of algorithms
- Understanding the hardware characteristics
- Interfacing with pre- and postprocessing tools

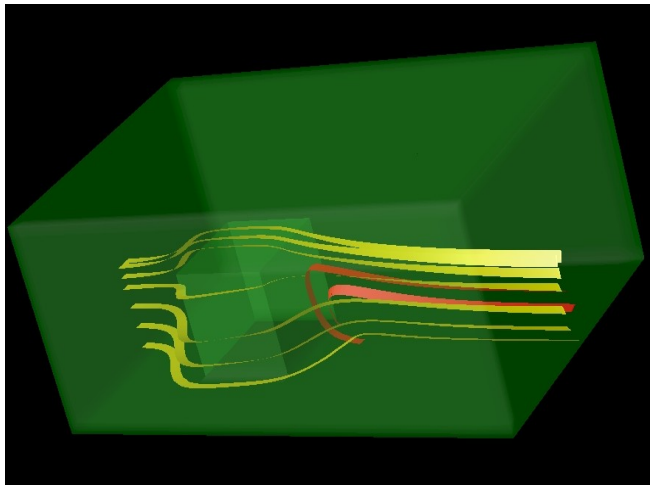
Together, these are called *High Performance Computing.*

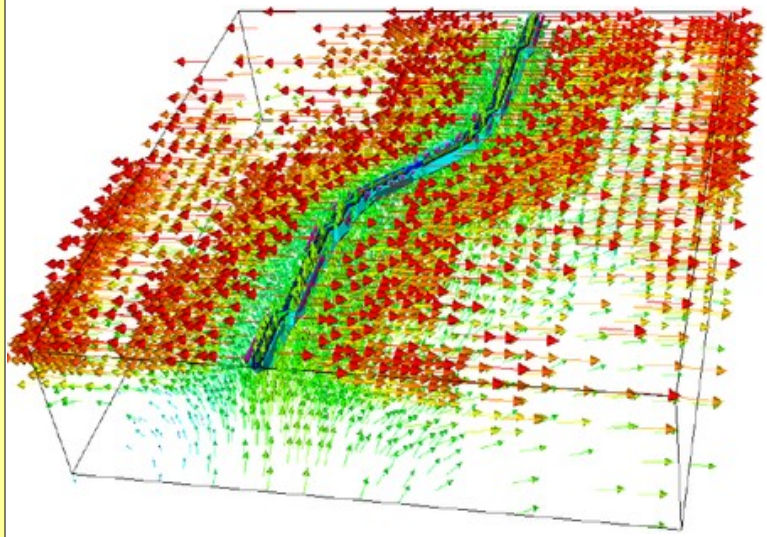
Examples of FEM applications in HPC

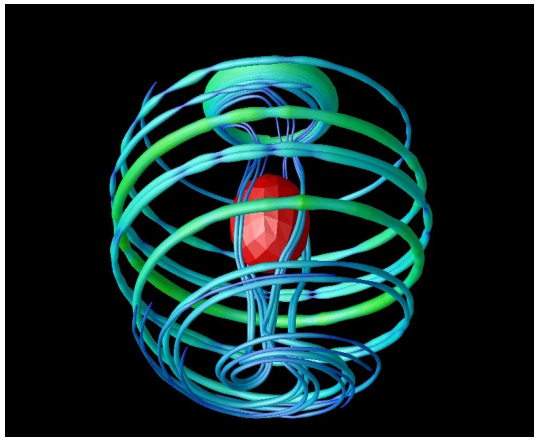
Examples from a wide variety of fields in my own work:

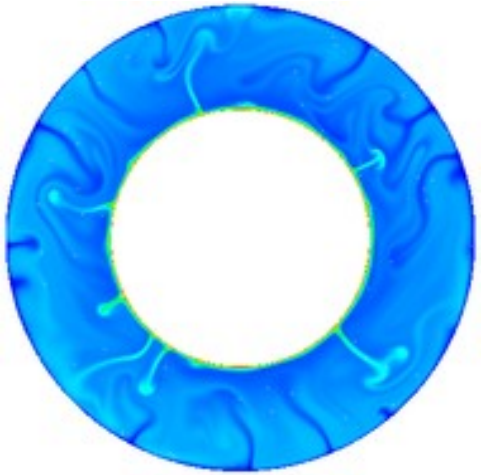






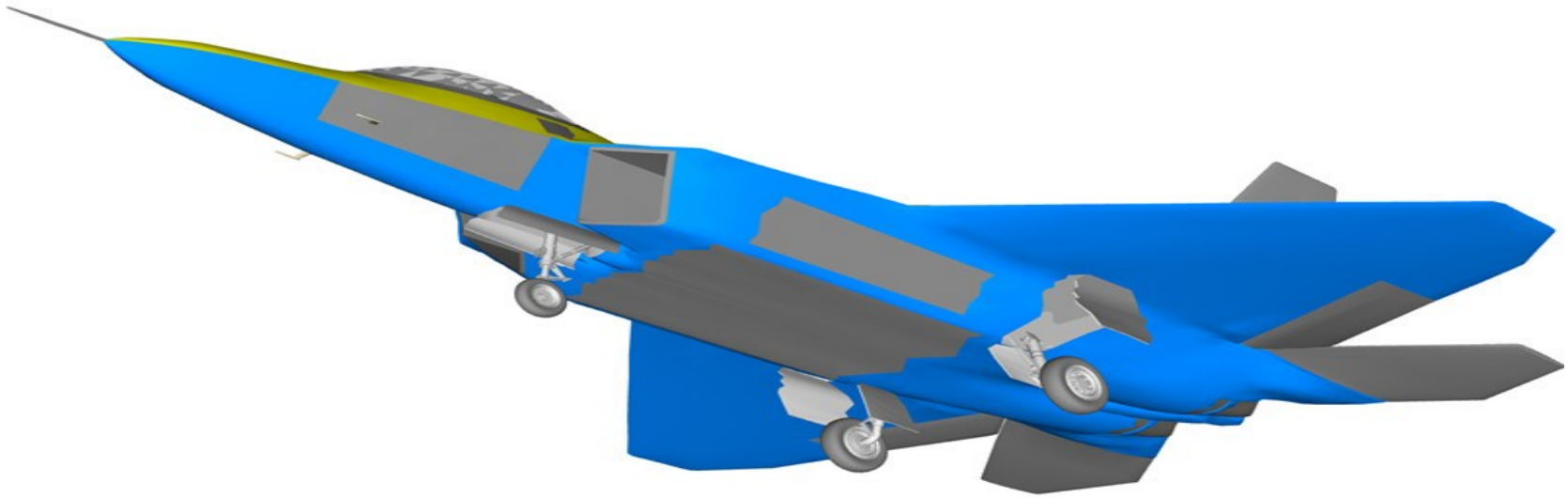






Workflow for HPC in PDEs

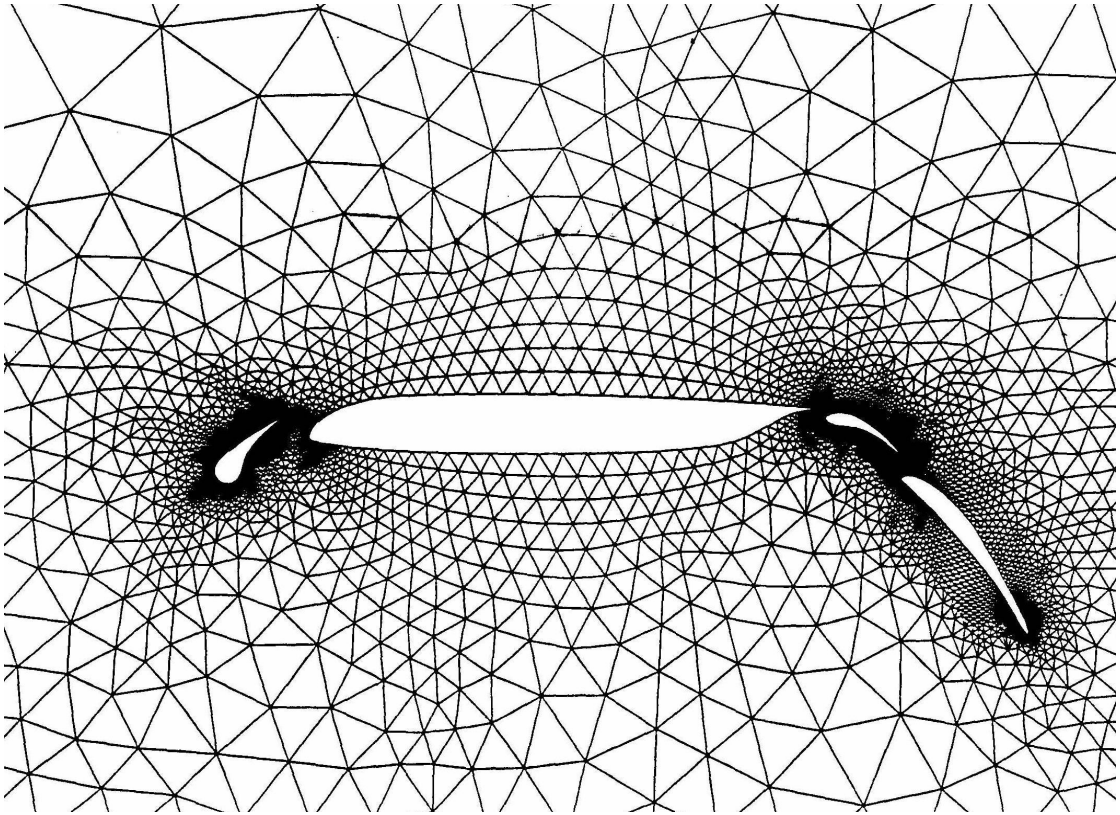
Step 1: Identify geometry and details of the model



May involve tens of thousands of pieces, very labor intensive, interface to designers and to manufacturing

Workflow for HPC in PDEs

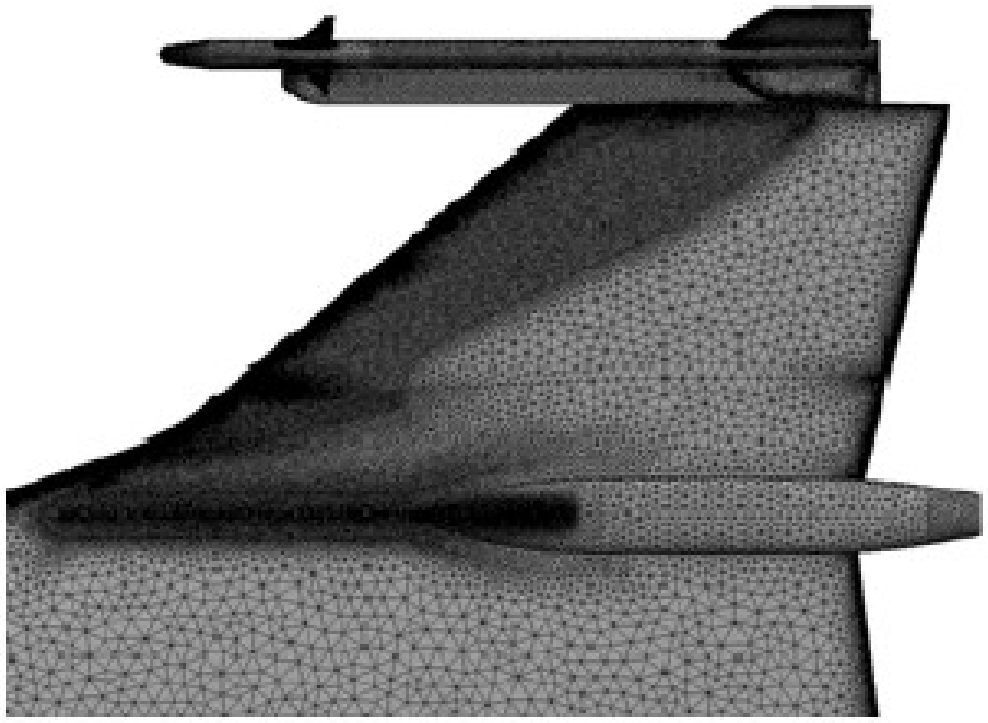
Step 2: Mesh generation and maybe partitioning (preprocessing)



May involve 10s of millions or more of cells; requires lots of memory; very difficult to parallelize

Workflow for HPC in PDEs

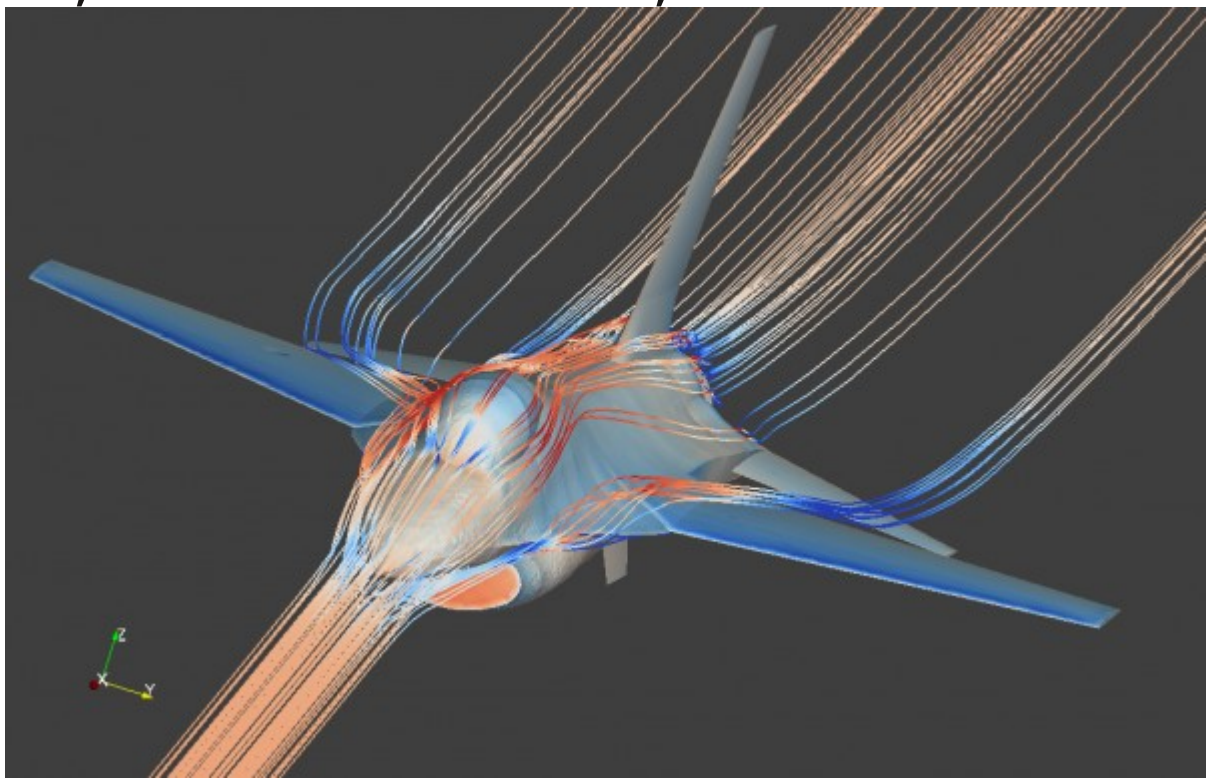
Step 2: Mesh generation and maybe partitioning (preprocessing)



May involve 10s of millions or more of cells; requires lots of memory; very difficult to parallelize

Workflow for HPC in PDEs

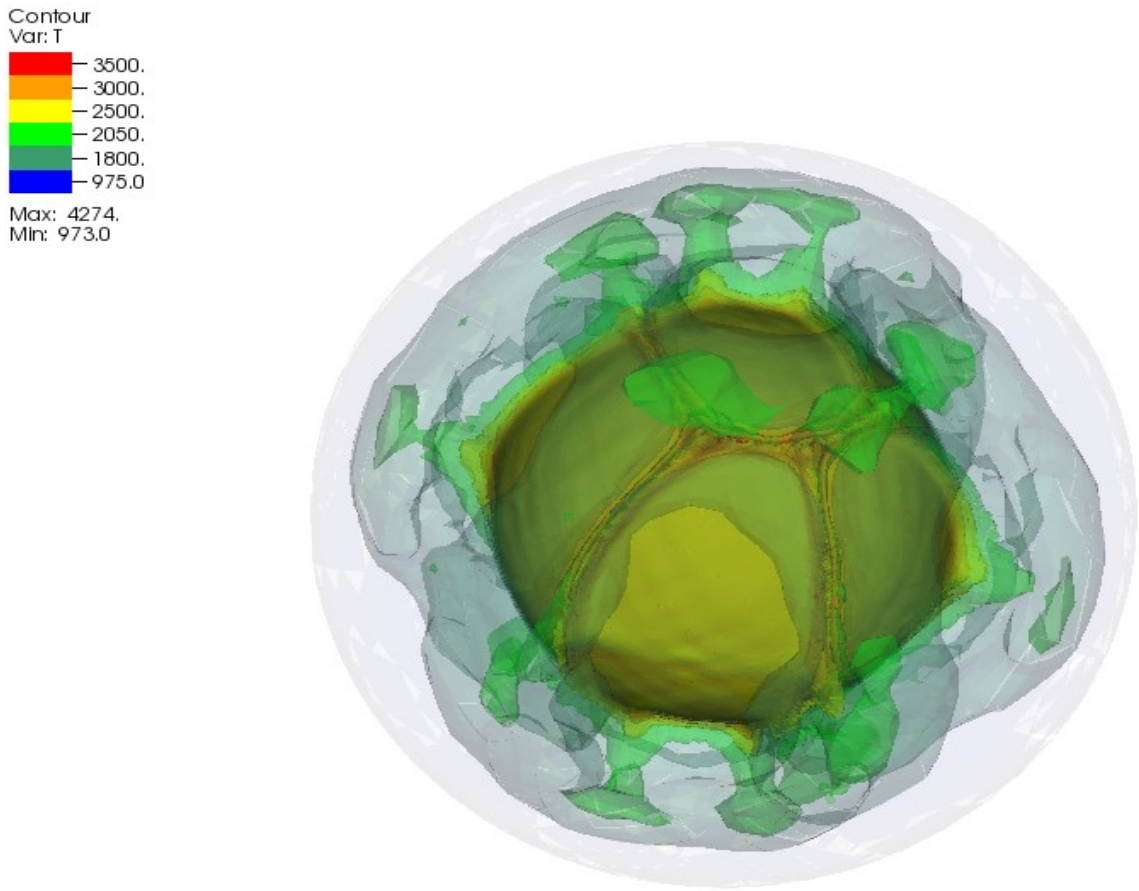
Step 3: Solve model on this mesh using finite elements, finite volumes, finite differences, …



Involves some of the biggest computations ever done, 10,000s of processors, millions of CPU hours, wide variety of algorithms

Workflow for HPC in PDEs

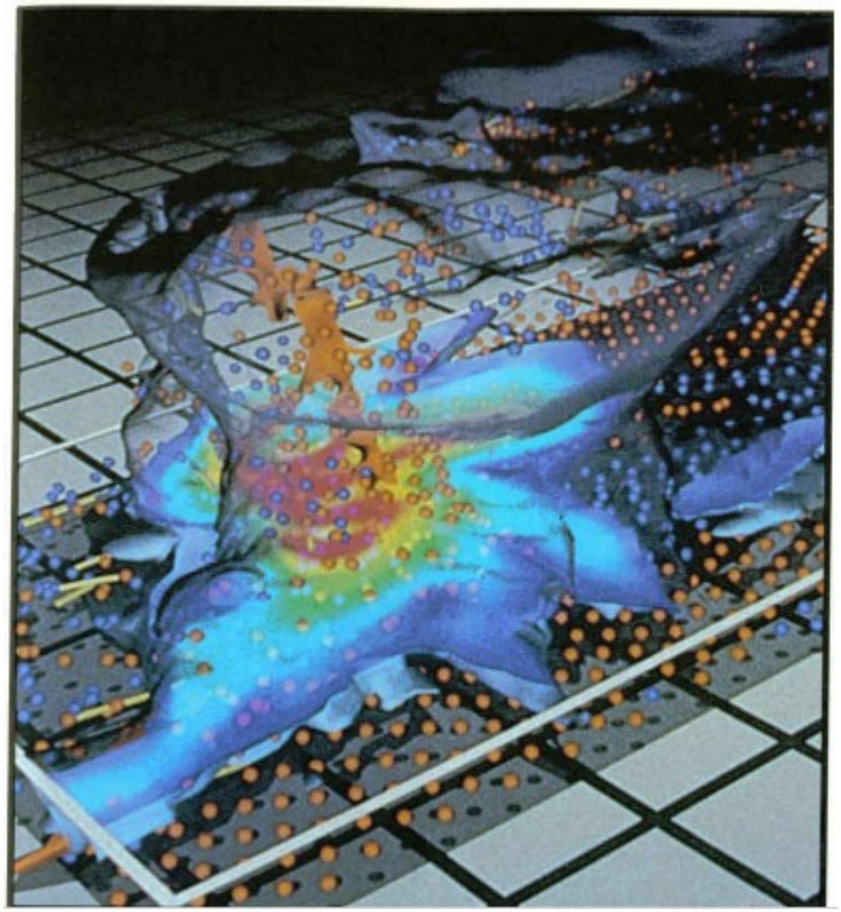
Step 4: Visualization to learn from the numerical results



Can be done in parallel, difficulty is amounts of data.

Workflow for HPC in PDEs

Step 4: Visualization to learn from the numerical results



Goal: Not to *plot data*, but to *provide insight*!

Workflow for HPC in PDEs

Step 5: Repeat

- To improve on the design
- To investigate different conditions (speed, altitude, angle of attack, …)
- To vary physical parameters that may not be known exactly
- To vary parameters of the numerical model (e.g. mesh size)
- To improve match with experiments

Workflow for HPC in PDEs

Each of these steps...

- Identify geometry and details of the model
- Preprocess: Mesh generation
- Solve problem with FEM/FVM/FDM
- Postprocess: Visualize
- Repeat

...needs software that requires:

- domain knowledge
- knowledge of the math. description of the problem
- knowledge of algorithm design
- knowledge of software design and management

Software issues in HPC

Ultimately, HPC is about *applications*, not just algorithms and their analysis.

Thus, we need to consider the issue of *software that implements* these applications:

- How complex is the software?
- How do we write software? Are there tools?
- How do we verify the correctness of the software?
- How do we validate the correctness of the model?

- Testing
- Documentation
- Social issues

Complexity of software

Many HPC applications are *several orders of magnitude* larger than everything you have probably ever seen!

For example, a crude measure of complexity is the number of lines of code in a package:

- Deal.II has 550k
- PETSc has 500k
- Trilinos has 3.1M

At this scale, software development does not work the same as for small projects:

- No single person has a global overview
- There are many years of work in such packages
- No person can remember even the code they wrote

Complexity of software

The only way to deal with the complexity of such software is to:

- *Modularize:* Different people are responsible for different parts of the project.
- *Define interfaces:* Only a small fraction of functions in a module is available to other modules
- *Document:* For users, for developers, for authors, and at different levels

- *Test, test, test*

How do we write software

Successful software must follow *the prime directive of software*:

- Developer time is the single most scarce resource!

As a consequence (part 1):

- Do not reinvent the wheel: use what others have already implemented (even if it's slower)
- Use the best tools (IDEs, graphical debuggers, graphical profilers, version control systems…)
- Do not make yourself the bottleneck (e.g. by not writing documentation)

- Delegate. You can't do it all.

How do we write software

Successful software must follow *the prime directive of software*:

- Developer time is the single most scarce resource!

As a consequence (part 2):

- Re-use code, don't duplicate
- Use strategies to *avoid* introducing bugs

- Test, test, test:
  - The earlier a bug is detected the easier it is to find
  - Even good programmers spend more time debugging code than writing it in the first place

Verification & validation (V&V): Verification

Verification refers to the process of ensuring that the software solves the problem it is supposed to solve:

"The program solves the problem correctly"

A common strategy to achieve this is to...

Verification & validation (V&V): Verification

Verification refers to the process of ensuring that the software solves the problem it is supposed to solve:

"The program solves the problem correctly"

A common strategy to achieve this is to *test test test:*

- *Unit tests* verify that a function/class does what it is supposed to do (assuming that correct result is known)
- *Integration tests* verify a whole algorithm (e.g. using what is known as the Method of Manufactured Solutions)
- Write *regression tests* that verify that the output of a program does not change over time

Software that is not tested does not produce the correct results!

(Note that I say "does not", and not "may not"!)

Verification & validation (V&V): Verification

Validation refers to the process of ensuring that the software solves a formulation that accurately represents the application:

"The program solves the correct problem"

The details of this go beyond this class.

Testing

Let me repeat the fundamental truth about software with more than a few 100 lines of code:

Software that is not tested does not produce the correct results!

No software that does not run lots of automatic tests can be good/usable.

As just one example:

- Deal.II runs ~2300 tests after every single change
- This takes ~10 CPU hours every time
- The test suite has another 250,000 lines of code.

Documentation

Documentation serves different purposes:

- It spells out to the developer what the *implementation* of a function/class is supposed to do (it's a *contract*)
- It tells a user what a function does
- It must come at different levels (e.g. functions, classes, modules, tutorial programs)

Also:

- Even in small projects, it reminds the author what she had in mind with a function after some time
- It avoids that everyone has to ask the developer for information (bottleneck!)
- Document the history of a code by using a version control system

Social issues

Most HPC software is a collaborative effort. Some of the most difficult aspects in HPC are of social nature:

- Can I modify this code?
- X just modified the code but didn't update the documentation and didn't write a test!
- Y1 has written a great piece of code but it doesn't conform to our coding style and he's unwilling to adjust it.
- Y2 seems clever but still has to learn. How do I interest her to collaborate without accepting subpar code?
- Z agreed to fix this bug 3 weeks ago but nothing has happened.
- M never replies to emails with questions about his code.