MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 22:

Some data structure design considerations

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 (Lack of encapsulation): You expose implementation details to the user:

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

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

Every piece of code using 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 (Unsuitable data structures): 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.

Implementation issues: Triangulations

Reason 3 (Inefficiency): 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;
    ...
};

Cell mesh[];
```

Observation: In loops over cells, we typically access like data on every cell!

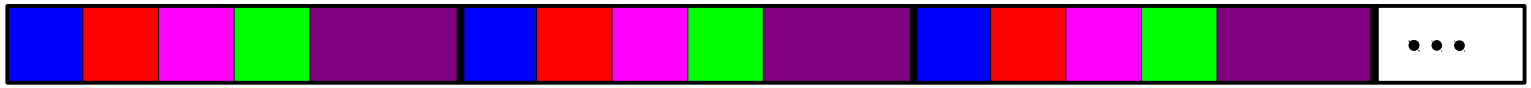
This is not an efficient arrangement of data in memory!

Implementation issues: Triangulations

Reason 3 (Inefficiency): 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;
    ...
};

Cell mesh[];
```

Memory layout (not to scale):



Members are consecutive in memory –> *cache misses*!

Implementation issues: Triangulations

A better approach (Separate the interface from the data structures): Identify which operations we need:

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

```
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

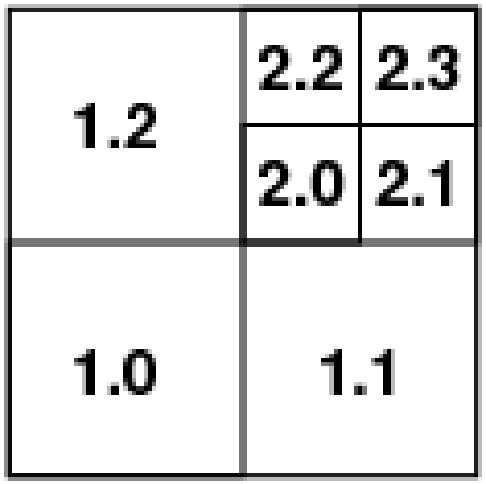
An example implementation:

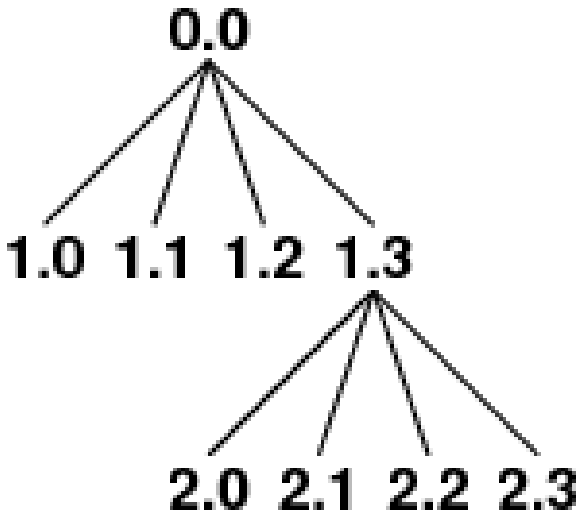
```
struct Triangulation  {
    cell_iterator begin () const { return cells.begin(); }
    ...
  private:
    std::list<Cell>     cells;
    std::vector<int>    cell_to_vertex_array;
    std::vector<Vertex> vertices;
};

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*!



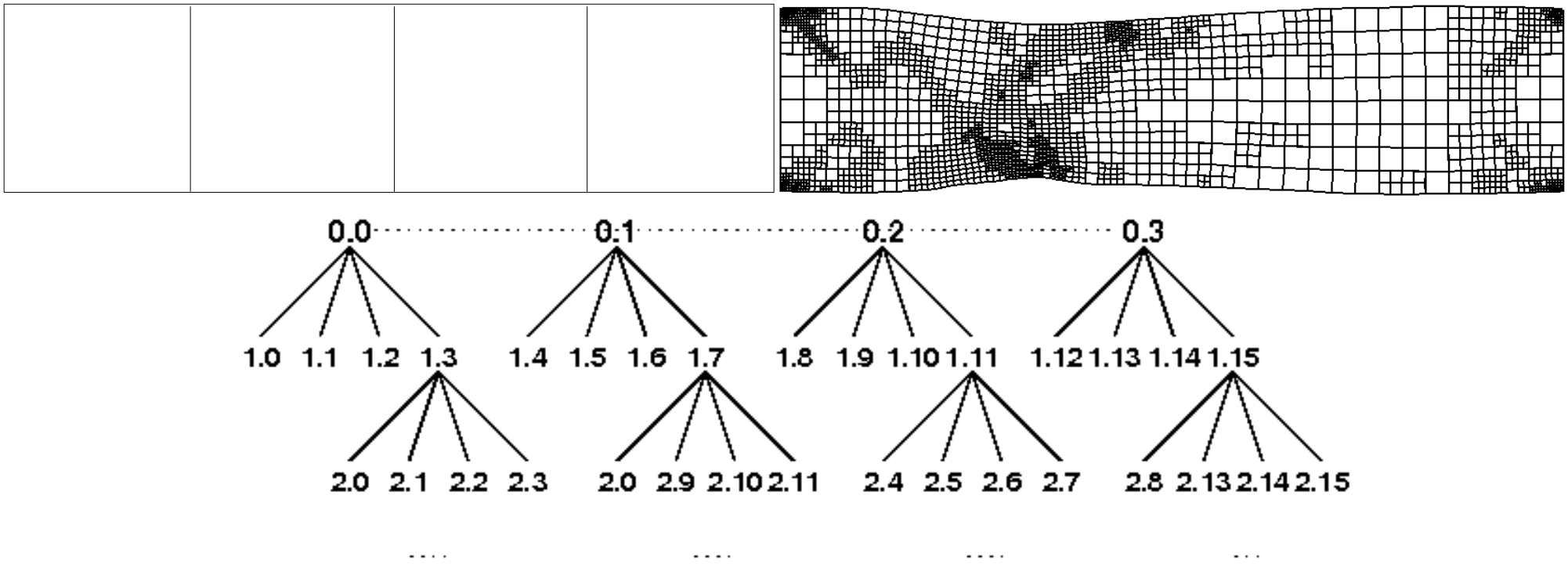


The 3d equivalent of this tree is an *octree*.

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 implementations:

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

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

class FELagrange       : public FiniteElement {…};
class FERaviartThomas  : public FiniteElement {…};
class FENedelec        : public FiniteElement {…};
… …
```

Implementation issues: Quadrature/Mapping

The same is true for quadrature objects:

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

class QGauss       : public Quadrature {…};
class QTrapezoidal : public Quadrature {…};
… …
```

Mapping classes are implemented similarly, providing linear, quadratic, ... Cartesian, C^1 mappings.

Implementation issues: FEValues objects

Remember that our integration procedure looked like this:

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

Note: This references

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

In practice, one never references

- Shape functions without mappings
- Mappings without 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:

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

Note:

- Some of these terms change from cell to cell

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

- Some are always the same (with the same shape functions and quadrature points):

$$\nabla\hat{\phi}_i(\hat{x}_q), \quad w_q$$

- 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 top of loop over cells, computes immutable components once (values, gradients on reference cell)
- Whenever we move to next cell, re-computes variable parts (things that depend on the location of vertices)
- Analyzes whether next cell is similar to previous one to save computations. E.g.:
  - If cell is translation of previous one, then Jacobian matrix is the same.
  - If translation + rotation, then determinant is the same

Implementation issues: Linear algebra

Appropriate data structures for vectors are obvious: Arrays.

For sparse matrices, one typically uses the *compressed sparse row* (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 entry 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)

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University