UFC - Unified Form-assembly Code

Martin Sandve Alnæs[1] Anders Logg[1] Kent-André Mardal[1]
Ola Skavhaug[1] Hans-Petter Langtangen[1]

Simula Research Laboratory[1]

8th of November 2006

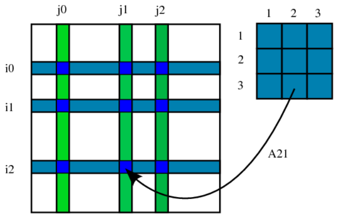
The main components of finite element software

- Defining the problem:
  - A weak form, $a(u, v) = c(v)$, f.ex. using FFC or SyFi
  - Finite element spaces (K,P,N), f.ex. using FIAT or SyFi
- Constructing a mesh Ω_h
- Assembling the matrices, vectors, or more generally tensors $A_i = a(v^1_{i_1}, v^2_{i_2}, \cdots, v^r_{i_r}), \quad \{v^j_k\} \in V^j_h$, f.ex. using Dolfin or PyCC
- Solving the linear system, $Au = b$

We want to define an interface to separate the assembly from the problem definition, to obtain interoperability between different libraries.

Assembling the tensor A_i with the Finite Element Method

- for each cell K in mesh Ω_h:
  - tabulate local to global mappings $\iota_K^j : [1, n_{loc}^j] \rightarrow [1, n_{glob}^j]$
  - tabulate local coefficient values $w_i^{K,j} = w_i^{\iota_K^j}$
  - compute element tensor A^K from K and $w_i^{K,j}$
  - add A_i^K to $A_{\iota_K(i)}$



Note that the code is simplified on these slides

For reasons of space on the slides:

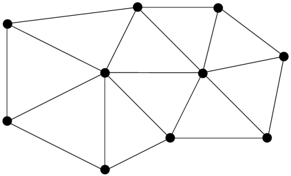
- Constructors and destructors are skipped
- Everything is public
- All functions presented are pure virtual

The latest draft of UFC can be found in the Mercurial repository at *http://fenics.org/hg/ufc/*.

We will need to know some properties of the global mesh

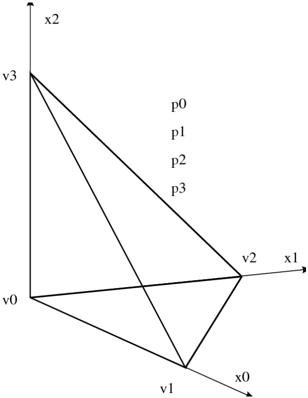
- A mesh entity is a vertex, edge, face or volume.
- num_entities[i] contains the total number of entities with topological dimension i in the mesh.

```
class mesh
{
  unsigned int topological_dimension;

  unsigned int geometric_dimension;

  unsigned int* num_entities;
};
```



A simple view of a mesh cell is also needed

```
class cell
{
  /// Array of global indices for
  /// the mesh entities of the cell
  unsigned int** entity_indices;

  /// Array of coordinates for the
  /// vertices of the cell
  double** coordinates;
};
```



The properties of an element tensor

An element tensor has rank r and a number of coefficient arguments n. Arguments after ; are input coefficients.

- $A_i = A_{(i_1,\cdots,i_r)} = a(v^1_{i_1}, \cdots, v^r_{i_r}; w_1, \cdots, w_n)$

In addition to $a(\cdots)$, we need the finite element space for each argument v_i and w_i.

Defining the argument spaces for $a(v_1, \cdots, v_r; w_1, \cdots, w_n)$

An element tensor can create a node map and finite element for each of its arguments.

```
class element_tensor
{
  unsigned int rank() const;

  unsigned int num_coefficients() const;

  node_map * create_node_map(unsigned int i) const;

  finite_element* create_finite_element(unsigned int i) const;

  /// ... more on the next slides
};
```

Computing the element tensor is the core of the assembly

- Boundary integrals are handled separately.
- Coefficients are passed as local nodal values in *w*.

```
class element_tensor
{
  bool has_interior_contribution() const;

  bool has_boundary_contribution() const;

  void tabulate_interior(double* A,
                         const double * const * w,
                         const cell& c) const;

  void tabulate_boundary(double* A,
                         const double * const * w,
                         const cell& c,
                         unsigned int facet) const;
};
```

Mapping from local to global degrees of freedom

Each node in the finite element space is usually associated with a mesh entity, but doesn't have to be.

```
class node_map
{
  unsigned int local_dimension() const;

  void tabulate_nodes(unsigned int *nodes, const mesh& m,
                      const cell& c) const;

  /// ... more on the next slides
};
```

The node map may depend on all the mesh cells

A node map can be initialized once for a mesh and cached for later use

```
class node_map
{
  bool needs_mesh_entities(unsigned int d) const;

  bool init_mesh(const mesh& mesh);

  void init_cell(const mesh& mesh, const cell& cell);

  unsigned int global_dimension() const;
};
```

The properties of a finite element

A finite element is a triplet (K, P, N), where

- K is the geometric cell
- P is a function space on K with a basis $\{\phi_i\}$
- $N = \{\nu_i\}$ is a basis for P', $\nu_i : P \rightarrow \Re$

$$u(x) = \sum_{i=1}^{n_{glob}} \nu_i(u)\phi_i(x)$$

Values in a function space can be general rank r tensors

$\phi_i(x)$ is often a scalar (pressure, temperature), vector (velocity, displacement) or matrix (stress tensor)

```
class finite_element
{
  unsigned int value_rank() const;

  unsigned int value_dimension(unsigned int i) const;
};
```

$\phi_i(x)$ and $\nu_i(f)$ can be evaluated directly

Interpolation at vertex values is commonly used for postprocessing

```
class finite_element
{
  /// Evaluate basis function i at the point
  /// x = (x[0], x[1], ...) in the cell
  void evaluate_basis(double* values, const double* x,
                      unsigned int i, const cell& c) const;

  /// Evaluate node i on the function f
  double evaluate_node(unsigned int i, const function& f,
                       const cell& c) const;

  /// Interpolate vertex values from nodal values
  void interpolate_vertex_values(double* vertex_values,
                                 const double* nodal_values) const;
};
```

A utility class for a general tensor valued function

Used for evaluation of a function in finite element nodes.

```
class function
{
  /// Evaluate the function at the point
  /// x = (x[0], x[1], ...) in the cell
  void evaluate(double* values, const double* x,
                const cell& c) const;
};
```

Mixed elements can be handled uniformly or separated

These two functions give optional access to subelements.

```
class finite_element
{
  unsigned int num_sub_elements(unsigned int i) const;

  const finite_element& sub_element(unsigned int i) const;
};
```

To sum it all up, here is a prototype assembly routine

```
for(; !cell_iter->end(); cell_iter->next()) {
  const ufc::cell & cell = cell_iter->get_cell();

  for(uint i=0; i<num_node_maps; i++) {
    node_maps[i]->tabulate_nodes(loc2glob[i],
                                 *umesh, cell);
  }

  for(uint i=0; i<num_coefficients; i++) {
    for(uint j=0; j<node_map_dim[i]; j++) {
      local_w[j] = global_w[ loc2glob[i][j] ];
    }
  }

  elm_tensor.tabulate_interior(Ae, w, cell);

  assembler.assemble(Ae, Ae_strides, rank,
                     loc2glob, node_map_dim);
}
```