. UFC: A Finite Element Code Generation Interface

Article · February 2012

DOI: 10.1007/978-3-642-23099-8_16

3 authors:

Martin Sandve Alnæs
Simula Research Laboratory
17 PUBLICATIONS **1,290** CITATIONS
SEE PROFILE

Anders Logg
Chalmers University of Technology
94 PUBLICATIONS **3,914** CITATIONS
SEE PROFILE

Kent-Andre Mardal
University of Oslo
233 PUBLICATIONS **4,976** CITATIONS
SEE PROFILE

1. UFC: A Finite Element Code Generation Interface

By Martin Sandve Alnæs, Anders Logg and Kent-Andre Mardal

A central component of FEniCS is the UFC interface (Unified Form-assembly Code). UFC is an interface between problem-specific and general-purpose components of finite element programs. In particular, the UFC interface defines the structure and signature of the code that is generated by the form compilers FFC and SFC for DOLFIN.

The UFC interface applies to a wide range of finite element problems (including mixed finite elements and discontinuous Galerkin methods) and may be used with libraries that differ widely in their design. For this purpose, the interface does not depend on any other FEniCS components (or other libraries) and consists only of a minimal set of abstract C++ classes using plain C arrays for data transfer.

This chapter gives a short overview of the UFC interface. For a more comprehensive discussion, we refer to the UFC manual [1] and the paper [2].

1.1 Finite element discretization and assembly

In Chapter [logg-3], we described the assembly algorithm for computing the global rank ρ tensor A corresponding to a multilinear form a of arity ρ,

$$\begin{aligned} &a : V_h^1 \times V_h^2 \times \cdots \times V_h^\rho \times W_h^1 \times W_h^2 \times \cdots \times W_h^n \to \mathbb{R}, \\ &a \mapsto a(v_1, v_2, \ldots, v_\rho; w_1, w_2, \ldots, w_n). \end{aligned} \tag{1.1}$$

Here, $\{V_h^j\}_{j=1}^\rho$ is a sequence of discrete function spaces for the *primary arguments* $\{v_j\}_{j=1}^\rho$ of the form and $\{W_h^j\}_{j=1}^n$ is a sequence of discrete function spaces for the *coefficients* $\{w_j\}_{j=1}^\rho$ of the form. Typically, the arity is $\rho = 1$ for a linear form or $\rho = 2$ for a bilinear form. In the simplest case, all function spaces are equal but there are many important examples, such as mixed methods, where the arguments come from different function spaces. The choice of coefficient function spaces depends on the application; a polynomial basis simplifies exact integration, while in some cases evaluating coefficients in quadrature points may be required.

As we saw in Chapter [logg-3], the global tensor A can be computed by summing contributions from the cells and facets of a mesh. We refer to these contributions as either cell tensors or facet tensors. Although one may formulate a generic assembly algorithm, the cell

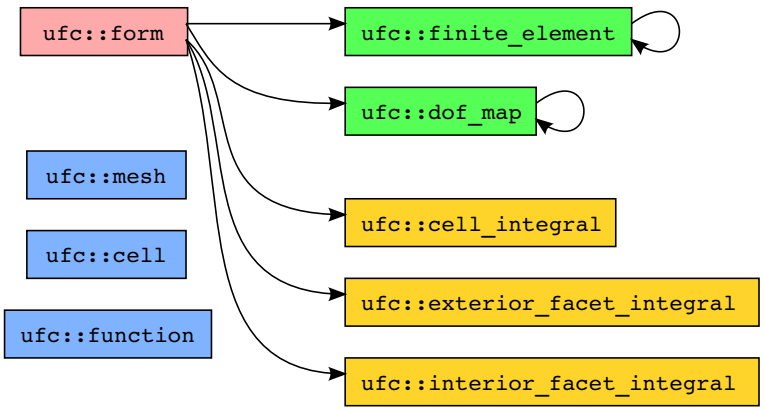

Figure 1.1: UML diagram of the UFC class relations

and facet tensors must be computed differently depending on the variational form, and their entries must be inserted differently into the global tensor depending on the choice of finite element spaces. This is handled in FEniCS by implementing a generic assembly algorithm (as part of DOLFIN) that relies on special-purpose generated code (by FFC or SFC) for computing the cell and facet tensors, and for computing the local-to-global map for insertion of the cell and facet tensors into the global matrix.

The UFC interface assumes that the multilinear form a in (1.1) can be expressed as a sum of integrals over the cells $\mathcal{T}$, the exterior facets $\partial_e\mathcal{T}$, and the interior facets $\partial_i\mathcal{T}$ of the mesh. The integrals may be expressed on disjoint subsets $\mathcal{T} = \cup_{k=1}^{n_c}\mathcal{T}_k$, $\partial_e\mathcal{T} = \cup_{k=1}^{n_e}\partial_e\mathcal{T}_k$, and $\partial_i\mathcal{T} = \cup_{k=1}^{n_i}\partial_i\mathcal{T}_k$ respectively. In particular, it is assumed that the multilinear form can be expressed in the following canonical form:

$$\begin{aligned} a(v_1, v_2, \ldots, v_\rho; w_1, w_2, \ldots, w_n) = &\sum_{k=1}^{n_c} \sum_{T\in\mathcal{T}_k} \int_T I_k^c(v_1, v_2, \ldots, v_\rho; w_1, w_2, \ldots w_n)\,\mathrm{d}x \\ &+ \sum_{k=1}^{n_e} \sum_{S\in\partial_e\mathcal{T}_k} \int_S I_k^e(v_1, v_2, \ldots, v_\rho; w_1, w_2, \ldots, w_n)\,\mathrm{d}s \\ &+ \sum_{k=1}^{n_i} \sum_{S^0\in\partial_i\mathcal{T}_k} \int_{S^0} I_k^i(v_1, v_2, \ldots, v_\rho; w_1, w_2, \ldots, w_n)\,\mathrm{d}s. \end{aligned} \tag{1.2}$$

We refer to an integral I_k^c over a cell T as a *cell integral*, an integral over an exterior facet S as an *exterior facet integral* (typically used to implement Neumann and Robin type boundary conditions), and to an integral over an interior facet S^0 as an *interior facet integral* (typically used in discontinuous Galerkin methods).

1.2 The UFC interface

The UFC interface consists of a small collection of abstract C++ classes that represent common components for assembling tensors using the finite element method. The full UFC interface is specified in a single header file `ufc.h`. The UFC classes are accompanied by a set of conventions for numbering of cell data and other arrays. Data is passed as plain C arrays

for efficiency and minimal dependencies. Most functions are declared `const`, reflecting that the operations they represent should not change the outcome of future operations.[1]

1.2.1 Class relations

Figure (1.1) shows all UFC classes and their relations. The classes `mesh`, `cell`, and `function` provide the means for communicating mesh and coefficient function data as arguments.

The integrals of (1.2) are represented by one of the following classes:

- `cell_integral`,
- `exterior_facet_integral`,
- `interior_facet_integral`.

Subclasses of `form` must implement factory functions which may be called to create integral objects. These objects in turn know how to compute their respective contribution from a cell or facet during assembly. A code fragment from the `form` class declaration is shown below.

```
class form
{
public:

  ...

  /// Create a new cell integral on sub domain i
  virtual cell_integral* create_cell_integral(unsigned int i) const = 0;

  /// Create a new exterior facet integral on sub domain i
  virtual exterior_facet_integral*
  create_exterior_facet_integral(unsigned int i) const = 0;

  /// Create a new interior facet integral on sub domain i
  virtual interior_facet_integral*
  create_interior_facet_integral(unsigned int i) const = 0;

};
```

The `form` class also specifies functions for creating `finite_element` and `dof_map` objects for the finite element function spaces $\{V_h^j\}_{j=1}^{\rho}$ and $\{W_h^j\}_{j=1}^{n}$ of the variational form. The `finite_element` object provides functionality such as evaluation of degrees of freedom and evaluation of basis functions and their derivatives. The `dof_map` object provides functionality such as tabulating the local-to-global map of degrees of freedom on a single element, as well as tabulation of subsets associated with particular mesh entities, used to apply Dirichlet boundary conditions and build connectivity information.

Both the `finite_element` and `dof_map` classes can represent mixed elements, in which case it is possible to obtain `finite_element` and `dof_map` objects for each subelement in a hierarchical manner. Vector elements composed of scalar elements are in this context seen as special cases of mixed elements where all subelements are equal. As an example, consider the `dof_map` for a $P_2 - P_1$ Taylor–Hood element. From this `dof_map` it is possible to extract one

[1] The exceptions are the functions to initialize a `dof_map`.

```
class cell
{
public:

  /// Constructor
  cell(): cell_shape(interval),
          topological_dimension(0), geometric_dimension(0),
          entity_indices(0), coordinates(0) {}

  /// Destructor
  virtual ~cell() {}

  /// Shape of the cell
  shape cell_shape;

  /// Topological dimension of the mesh
  unsigned int topological_dimension;

  /// Geometric dimension of the mesh
  unsigned int geometric_dimension;

  /// 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;

};
```

Figure 1.2: Data structure for communicating cell data.

`dof_map` for the quadratic vector element and one `dof_map` for the linear scalar element. From the vector element, a `dof_map` for the quadratic scalar element of each vector component can be obtained. This can be used to access subcomponents from the solution of a mixed system.

1.2.2 Stages in the assembly algorithm

Next, we focus on a few key parts of the interface and explain how these can be used to implement the assembly algorithm presented in Chapter [logg-3]. The general algorithm consists of three stages: (i) assembling the contributions from all cells, (ii) assembling the contributions from all exterior facets, and (iii) assembling the contributions from all interior facets.

Each of the three assembly stages (i)–(iii) is further composed of five steps. In the first step, a cell T is fetched from the mesh, typically implemented by filling a `cell` structure (see Figure 1.2) with coordinate data and global numbering of the mesh entities in the cell. This step depends on the specific mesh being used.

In the second step, the coefficients in $\{W_h^j\}_{j=1}^n$ are restricted to the local cell T. If a coefficient w_j is not given as a linear combination of basis functions for W_h^j, it must at this step be interpolated into W_h^j, using the interpolant defined by the degrees of freedom of W_h^j (one common choice of interpolation is point evaluation at the set of nodal points). In this case,

the coefficient function is passed as an implementation of the `function` interface (a simple functor) to the function `evaluate_dofs` in the UFC `finite_element` class.

```
/// Evaluate linear functionals for all dofs on the function f
virtual void evaluate_dofs(double* values,
                           const function& f,
                           const cell& c) const = 0;
```

In the third step, the local-to-global map of degrees of freedom is tabulated for each of the function spaces. That is, for each of the local discrete finite element spaces on T, we tabulate the corresponding global degrees of freedom.

```
/// Tabulate the local-to-global mapping of dofs on a cell
void dof_map::tabulate_dofs(unsigned int* dofs,
                            const mesh& m,
                            const cell& c) const
```

Here, `unsigned int* dofs` is a pointer to the first element of an array of unsigned integers that will be filled with the local-to-global map on the current cell during the function call.

In the fourth step, the local element tensor contributions (cell or exterior/interior facet tensors) are computed. This is done by a call to the function `tabulate_tensor`, illustrated below for a cell integral.

```
/// Tabulate the tensor for the contribution from a local cell
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const cell& c) const = 0;
```

Similarly, one may evaluate interior and exterior facet contributions using slightly different function signatures.

Finally, at the fifth step, the local element tensor contributions are added to the global tensor, using the local-to-global maps previously obtained by calls to the `tabulate_dofs` function. This is an operation that depends on the linear algebra backend used to store the global tensor.

1.2.3 Code generation utilities

UFC provides a number of utilities that can be used by form compilers to simplify the code generation process, including templates for creating subclasses of UFC classes and utilities for just-in-time compilation. These are distributed as part of the `ufc_utils` Python module.

Templates are available for all UFC classes listed in Figure 1.1 and consist of format strings for the skeleton of each subclass. The following code illustrates how to generate a subclass of the UFC `form` class.

```
from ufc_utils import form_combined

implementation = {}
implementation["signature"] = "return \"my form\""
implementation["rank"] = "return 2;"
```

```
implementation["num_coefficients"] = "return 0;"
...

print form % implementation
```

This generates code for a single header file that also contains the implementation of each function in the UFC `form` interface. It is also possible to generate code for separate header (`.h`) and implementation (`.cpp`) files by using the `form_header` and `form_implementation` templates.

The `ufc_utils` module also contains the utility function `build_ufc_module` that can be called to build a Python module based on generated UFC code. This process involves compilation, linking, and loading of the generated C++ code as well as generating a Python wrapper module using Instant/SWIG as described in Chapter [wilbers].

1.3 Examples

In this section, we demonstrate how UFC is used in practice for assembly of finite element forms. First, we demonstrate how one may implement a simple assembler based on generated UFC code. We then show examples of input to the form compilers FFC and SFC as well as part of the corresponding UFC code generated as output.

1.3.1 Assembler

Below, we include a sketch of a UFC-based implementation of the assembly of the global tensor A by summing the local contributions from all cells. The contributions from all exterior and interior facets may be computed similarly.

The implementation is incomplete and system specific details such as interaction with mesh and linear algebra libraries have been omitted. A full implementation of the assembly algorithm is available as part of DOLFIN. Part of this implementation is shown in Chapter [logg-3]. The DOLFIN implementation is similar to the implementation below but relies on DOLFIN-specific wrappers for the UFC data structures.

```
void assemble(..., ufc::form& form, ...)
{
  ...

  // Initialize mesh data structure
  ufc::mesh mesh;
  mesh.num_entities = new unsigned int[...];
  ...

  // Initialize cell data structure
  ufc::cell cell;
  cell.entity_indices = new unsigned int[...];
  cell.coordinates = new double[...];
  ...

  // Create cell integrals
  ufc::cell_integral** cell_integrals;
  cell_integrals = new ufc::cell_integral*[form.num_cell_integrals()];
```

```
for (unsigned int i = 0; i < form.num_cell_integrals(); i++)
  cell_integrals[i] = form.create_cell_integral(i);

// Create dofmaps
ufc::dof_maps** dof_maps;
dof_maps = new ufc::dof_map*[form.rank() + form.num_coefficients()];
for (unsigned int i = 0; i < form.rank() + form.num_coefficients(); i++)
{
  dof_maps[i] = form.create_dof_map(i);

  // Initialize dofmap
  if (dof_maps[i]->init_mesh(mesh))
  {
    // Iterate over cells
    for (...)
    {
      // Update cell data structure to current cell
      cell.entity_indices[...] = ...
      cell.coordinates[...] = ...
      ...

      // Initialize dofmap for cell
      dof_maps[i]->init_cell(mesh, cell);
    }

    dof_map.init_cell_finalize();
  }
}

// Initialize array of values for the cell tensor
unsigned int size = 1;
for (unsigned int i = 0; i < form.rank(); i++)
  size *= dof_maps[i]->max_local_dimension();
double* A_T = new double[size];

// Initialize array of local to global dofmaps
unsigned int** dofs = new unsigned int*[form.rank()];
for (unsigned int i = 0; i < form.rank(); i++)
  dofs[i] = new unsigned int[dof_maps[i]->max_local_dimension()];

// Initialize array of coefficient values
double** w = new double*[form.num_coefficients()];
for (unsigned int i = 0; i < form.num_coefficients(); i++)
  w[i] = new double[dof_maps[form.rank() + i]->max_local_dimension()];

// Iterate over cells
for (...)
{
  // Get number of subdomain for current cell
  const unsigned int sub_domain = ...

  // Update cell data structure to current cell
  cell.entity_indices[...] = ...
  cell.coordinates[...] = ...
  ...

  // Interpolate coefficients (library specific so omitted here)
```

```
    ...

    // Tabulate dofs for each dimension
    for (unsigned int i = 0; i < ufc.form.rank(); i++)
      dof_maps[i]->tabulate_dofs(dofs[i], mesh, cell);

    // Tabulate cell tensor
    cell_integrals[sub_domain]->tabulate_tensor(A_T, w, cell);

    // Add entries to global tensor (library specific so omitted here)
    ...
  }

  // Delete data structures
  delete [] mesh.num_entities;
  ...
}
```

1.3.2 *Generated UFC code*

The form language UFL described in Chapter [alnes-1] provides a simple language for specification of variational forms, which may be entered either directly in Python or in text files given to a form compiler. We consider the following definition of the bilinear form $a(v, u) = \langle \nabla v, \nabla u \rangle$ in UFL:

```
element = FiniteElement("CG", "triangle", 1)

v = TestFunction(element)
u = TrialFunction(element)
f = Coefficient(element)

a = inner(grad(v), grad(u))*dx
```

When compiling this code, a C++ header file is created, containing UFC code that may be used to assemble the global sparse stiffness matrix for Poisson's equation. Below, we present the code generated for evaluation of the element stiffness matrix for the bilinear form a using FFC. Similar code may be generated using SFC.

```
/// Tabulate the tensor for the contribution from a local cell
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const ufc::cell& c) const
{
  // Number of operations (multiply-add pairs) for Jacobian data:      11
  // Number of operations (multiply-add pairs) for geometry tensor:    8
  // Number of operations (multiply-add pairs) for tensor contraction: 11
  // Total number of operations (multiply-add pairs):                  30

  // Extract vertex coordinates
  const double * const * x = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = x[1][0] - x[0][0];
  const double J_01 = x[2][0] - x[0][0];
```

```
  const double J_10 = x[1][1] - x[0][1];
  const double J_11 = x[2][1] - x[0][1];

  // Compute determinant of Jacobian
  double detJ = J_00*J_11 - J_01*J_10;

  // Compute inverse of Jacobian
  const double K_00 =  J_11 / detJ;
  const double K_01 = -J_01 / detJ;
  const double K_10 = -J_10 / detJ;
  const double K_11 =  J_00 / detJ;

  // Set scale factor
  const double det = std::abs(detJ);

  // Compute geometry tensor
  const double G0_0_0 = det*(K_00*K_00 + K_01*K_01);
  const double G0_0_1 = det*(K_00*K_10 + K_01*K_11);
  const double G0_1_0 = det*(K_10*K_00 + K_11*K_01);
  const double G0_1_1 = det*(K_10*K_10 + K_11*K_11);

  // Compute element tensor
  A[0] = 0.500000000000000*G0_0_0 + 0.500000000000000*G0_0_1
       + 0.500000000000000*G0_1_0 + 0.500000000000000*G0_1_1;
  A[1] = -0.500000000000000*G0_0_0 - 0.500000000000000*G0_1_0;
  A[2] = -0.500000000000000*G0_0_1 - 0.500000000000000*G0_1_1;
  A[3] = -0.500000000000000*G0_0_0 - 0.500000000000000*G0_0_1;
  A[4] = 0.500000000000000*G0_0_0;
  A[5] = 0.500000000000000*G0_0_1;
  A[6] = -0.500000000000000*G0_1_0 - 0.500000000000000*G0_1_1;
  A[7] = 0.500000000000000*G0_1_0;
  A[8] = 0.500000000000000*G0_1_1;
}
```

Having computed the element tensor, one needs to compute the local-to-global map in order to know where to insert the local contributions in the global tensor. This map may be obtained by calling the member function `tabulate_dofs` of the class `dof_map`. FFC uses an implicit ordering scheme, based on the indices of the topological entities in the mesh. This information may be extracted from the `cell` attribute `entity_indices`. For linear Lagrange elements on triangles, each degree of freedom is associated with a global vertex. Hence, FFC constructs the map by picking the corresponding global vertex number for each degree of freedom as demonstrated below.

```
virtual void tabulate_dofs(unsigned int* dofs,
                           const ufc::mesh& m,
                           const ufc::cell& c) const
{
  dofs[0] = c.entity_indices[0][0];
  dofs[1] = c.entity_indices[0][1];
  dofs[2] = c.entity_indices[0][2];
}
```

For quadratic Lagrange elements, a similar map is generated based on global vertex and edge numbers (entities of dimension zero and one respectively). We list the code for `tabulate_dofs` generated by FFC for quadratic Lagrange elements below.

Reference cell	Dimension	#Vertices	#Facets
The reference interval	1	2	2
The reference triangle	2	3	3
The reference quadrilateral	2	4	4
The reference tetrahedron	3	4	4
The reference hexahedron	3	8	6

Table 1.1: Reference cells covered by the UFC specification.

```
virtual void tabulate_dofs(unsigned int* dofs,
                           const ufc::mesh& m,
                           const ufc::cell& c) const
{
  unsigned int offset = 0;
  dofs[0] = offset + c.entity_indices[0][0];
  dofs[1] = offset + c.entity_indices[0][1];
  dofs[2] = offset + c.entity_indices[0][2];
  offset += m.num_entities[0];
  dofs[3] = offset + c.entity_indices[1][0];
  dofs[4] = offset + c.entity_indices[1][1];
  dofs[5] = offset + c.entity_indices[1][2];
  offset += m.num_entities[1];
}
```

1.4 Numbering conventions

UFC relies on a set of numbering conventions for cells, vertices, and other mesh entities. The numbering scheme ensures that form compilers (FFC and SFC) and assemblers (DOLFIN) can communicate data required for tabulating the cell and facet tensors as well as local-to-global maps.

1.4.1 Reference cells

The following five reference cells are covered by the UFC specification: the reference *interval*, the reference *triangle*, the reference *quadrilateral*, the reference *tetrahedron*, and the reference *hexahedron* (see Table 1.1). The UFC specification assumes that each cell in a finite element mesh is always isomorphic to one of the reference cells.

The reference interval

The reference interval is shown in Figure 1.3 and is defined by its two vertices with coordinates as specified in Table 1.2.

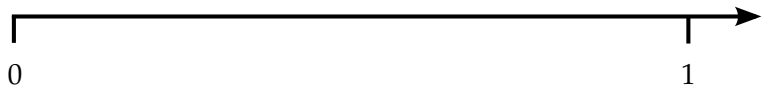

Figure 1.3: The reference interval.

Vertex	Coordinate
v_0	$x = 0$
v_1	$x = 1$

Table 1.2: Vertex coordinates of the reference interval.

The reference triangle

The reference triangle is shown in Figure 1.4 and is defined by its three vertices with coordinates as specified in Table 1.3.

The reference quadrilateral

The reference quadrilateral is shown in Figure 1.5 and is defined by its four vertices with coordinates as specified in Table 1.4.

The reference tetrahedron

The reference tetrahedron is shown in Figure 1.6 and is defined by its four vertices with coordinates as specified in Table 1.5.

The reference hexahedron

The reference hexahedron is shown in Figure 1.7 and is defined by its eight vertices with coordinates as specified in Table 1.6.

1.4.2 Numbering of mesh entities

The UFC specification dictates a certain numbering of the vertices, edges etc. of the cells of a finite element mesh. First, an *ad hoc* numbering is picked for the vertices of each cell. Then, the remaining entities are ordered based on a simple rule, as described in detail below.

Vertex	Coordinate
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (0, 1)$

Table 1.3: Vertex coordinates of the reference triangle.

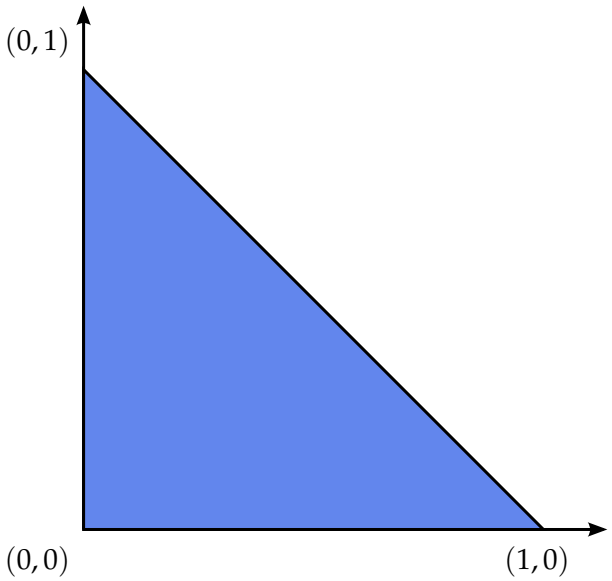

Figure 1.4: The reference triangle.

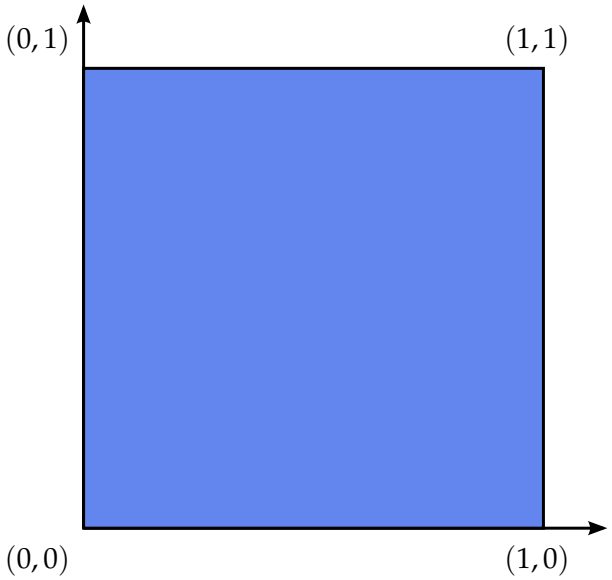

Figure 1.5: The reference quadrilateral.

Vertex	Coordinate
v_0	$x = (0, 0)$
v_1	$x = (1, 0)$
v_2	$x = (1, 1)$
v_3	$x = (0, 1)$

Table 1.4: Vertex coordinates of the reference quadrilateral.

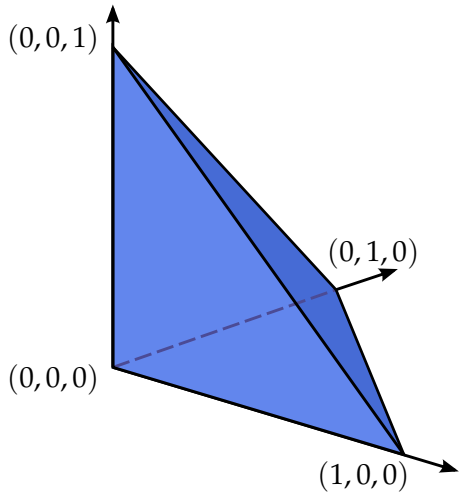

Figure 1.6: The reference tetrahedron.

Vertex	Coordinate
v_0	$x = (0, 0, 0)$
v_1	$x = (1, 0, 0)$
v_2	$x = (0, 1, 0)$
v_3	$x = (0, 0, 1)$

Table 1.5: Vertex coordinates of the reference tetrahedron.

Vertex	Coordinate	Vertex	Coordinate
v_0	$x = (0, 0, 0)$	v_4	$x = (0, 0, 1)$
v_1	$x = (1, 0, 0)$	v_5	$x = (1, 0, 1)$
v_2	$x = (1, 1, 0)$	v_6	$x = (1, 1, 1)$
v_3	$x = (0, 1, 0)$	v_7	$x = (0, 1, 1)$

Table 1.6: Vertex coordinates of the reference hexahedron.

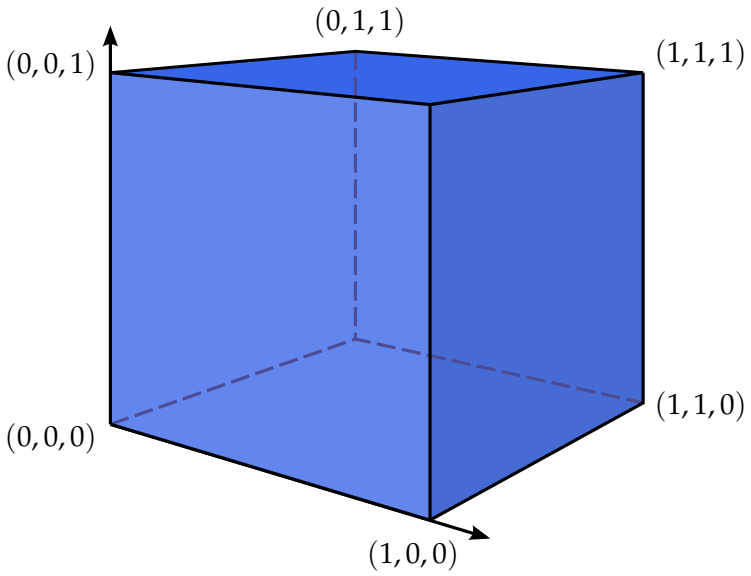

Figure 1.7: The reference hexahedron.

Entity	Dimension	Codimension
Vertex	0	–
Edge	1	–
Face	2	–
Facet	–	1
Cell	–	0

Table 1.7: Named mesh entities.

Basic concepts

The topological entities of a cell (or mesh) are referred to as *mesh entities*. A mesh entity can be identified by a pair (d, i), where d is the topological dimension of the mesh entity and i is a unique index of the mesh entity. Mesh entities are numbered within each topological dimension from 0 to $n_d - 1$, where n_d is the number of mesh entities of topological dimension d.

For convenience, mesh entities of topological dimension 0 are referred to as *vertices*, entities of dimension 1 as *edges*, entities of dimension 2 as *faces*, entities of *codimension* 1 as *facets*, and entities of codimension 0 as *cells*. These concepts are summarized in Table 1.7.

Thus, the vertices of a tetrahedron are identified as $v_0 = (0, 0)$, $v_1 = (0, 1)$, and $v_2 = (0, 2)$, the edges are $e_0 = (1, 0)$, $e_1 = (1, 1)$, $e_2 = (1, 2)$, $e_3 = (1, 3)$, $e_4 = (1, 4)$, and $e_5 = (1, 5)$, the faces (facets) are $f_0 = (2, 0)$, $f_1 = (2, 1)$, $f_2 = (2, 2)$, and $f_3 = (2, 3)$, and the cell itself is $c_0 = (3, 0)$.

Numbering of vertices

For simplicial cells (intervals, triangles, and tetrahedra) of a finite element mesh, the vertices are numbered locally based on the corresponding global vertex numbers. In particular, a tuple of increasing local vertex numbers corresponds to a tuple of increasing global vertex numbers. This is illustrated in Figure 1.8 for a mesh consisting of two triangles.

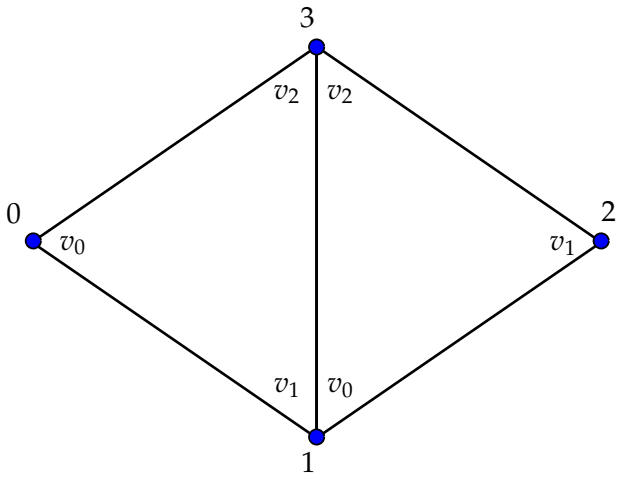

Figure 1.8: The vertices of a simplicial mesh are numbered locally based on the corresponding global vertex numbers.

For non-simplicial cells (quadrilaterals and hexahedra), the numbering is arbitrary, as long as each cell is topologically isomorphic to the corresponding reference cell by matching each vertex with the corresponding vertex in the reference cell. This is illustrated in Figure 1.9 for a mesh consisting of two quadrilaterals.

Numbering of other mesh entities

When the vertices have been numbered, the remaining mesh entities are numbered within each topological dimension based on a *lexicographical ordering* of the corresponding ordered tuples of *non-incident vertices*.

As an illustration, consider the numbering of edges (the mesh entities of topological dimension one) on the reference triangle in Figure 1.10. To number the edges of the reference triangle, we identify for each edge the corresponding non-incident vertices. For each edge, there is only one such vertex (the vertex opposite to the edge). We thus identify the three edges in the reference triangle with the tuples (v_0), (v_1), and (v_2). The first of these is edge e_0 between vertices v_1 and v_2 opposite to vertex v_0, the second is edge e_1 between vertices v_0 and v_2 opposite to vertex v_1, and the third is edge e_2 between vertices v_0 and v_1 opposite to vertex v_2.

Similarly, we identify the six edges of the reference tetrahedron with the corresponding non-incident tuples (v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_1, v_3), and (v_2, v_3). The first of these is edge e_0 between vertices v_2 and v_3 opposite to vertices v_0 and v_1 as shown in Figure 1.11.

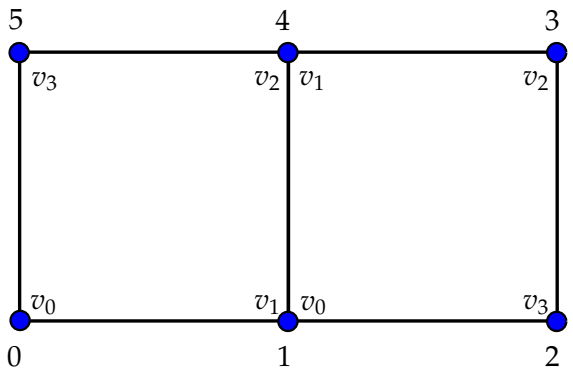

Figure 1.9: The local numbering of vertices of a non-simplicial mesh is arbitrary, as long as each cell is topologically isomorphic to the reference cell by matching each vertex to the corresponding vertex of the reference cell.

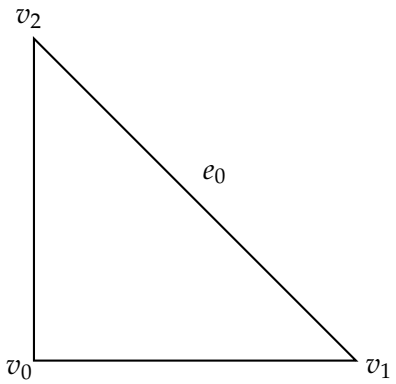

Figure 1.10: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertex v_0.

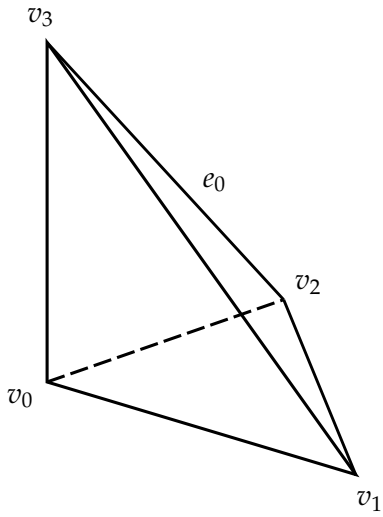

Figure 1.11: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertices v_0 and v_1.

Relative ordering

The relative ordering of mesh entities with respect to other incident mesh entities follows by sorting the entities by their (global) indices. Thus, the pair of vertices incident to the first edge e_0 of a triangular cell is (v_1, v_2), not (v_2, v_1). Similarly, the first face f_0 of a tetrahedral cell is incident to vertices (v_1, v_2, v_3).

For simplicial cells, the relative ordering in combination with the convention of numbering the vertices locally based on global vertex indices means that two incident cells will always agree on the orientation of incident subsimplices. Thus, two incident triangles will agree on the orientation of the common edge and two incident tetrahedra will agree on the orientation of the common edge(s) and the orientation of the common face (if any). This is illustrated in Figure 1.12 for two incident triangles sharing a common edge.

Limitations

The UFC specification is only concerned with the ordering of mesh entities with respect to entities of larger topological dimension. In other words, the UFC specification is only concerned with the ordering of incidence relations of the class $d - d'$ where $d > d'$. For example, the UFC specification is not concerned with the ordering of incidence relations of the class $0 - 1$, that is, the ordering of edges incident to vertices.

Numbering schemes for reference cells

The numbering scheme is demonstrated below for cells isomorphic to each of the five reference cells.

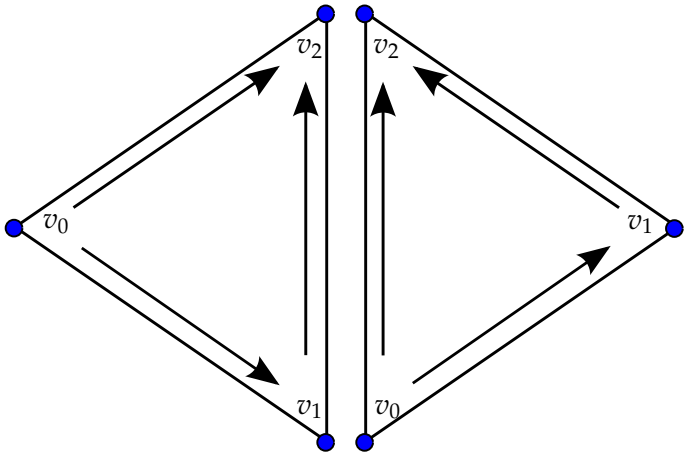

Figure 1.12: Two incident triangles will always agree on the orientation of the common edge.

Numbering of mesh entities on intervals

Entity	Incident vertices	Non-incident vertices
$v_0 = (0,0)$	(v_0)	(v_1)
$v_1 = (0,1)$	(v_1)	(v_0)
$c_0 = (1,0)$	(v_0, v_1)	$\emptyset$

Numbering of mesh entities on triangular cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0,0)$	(v_0)	(v_1, v_2)
$v_1 = (0,1)$	(v_1)	(v_0, v_2)
$v_2 = (0,2)$	(v_2)	(v_0, v_1)
$e_0 = (1,0)$	(v_1, v_2)	(v_0)
$e_1 = (1,1)$	(v_0, v_2)	(v_1)
$e_2 = (1,2)$	(v_0, v_1)	(v_2)
$c_0 = (2,0)$	(v_0, v_1, v_2)	$\emptyset$

Numbering of mesh entities on quadrilateral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0,0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0,1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0,2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0,3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1,0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1,1)$	(v_1, v_2)	(v_0, v_3)
$e_2 = (1,2)$	(v_0, v_3)	(v_1, v_2)
$e_3 = (1,3)$	(v_0, v_1)	(v_2, v_3)
$c_0 = (2,0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

Numbering of mesh entities on tetrahedral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0,0)$	(v_0)	(v_1, v_2, v_3)
$v_1 = (0,1)$	(v_1)	(v_0, v_2, v_3)
$v_2 = (0,2)$	(v_2)	(v_0, v_1, v_3)
$v_3 = (0,3)$	(v_3)	(v_0, v_1, v_2)
$e_0 = (1,0)$	(v_2, v_3)	(v_0, v_1)
$e_1 = (1,1)$	(v_1, v_3)	(v_0, v_2)
$e_2 = (1,2)$	(v_1, v_2)	(v_0, v_3)
$e_3 = (1,3)$	(v_0, v_3)	(v_1, v_2)
$e_4 = (1,4)$	(v_0, v_2)	(v_1, v_3)
$e_5 = (1,5)$	(v_0, v_1)	(v_2, v_3)
$f_0 = (2,0)$	(v_1, v_2, v_3)	(v_0)
$f_1 = (2,1)$	(v_0, v_2, v_3)	(v_1)
$f_2 = (2,2)$	(v_0, v_1, v_3)	(v_2)
$f_3 = (2,3)$	(v_0, v_1, v_2)	(v_3)
$c_0 = (3,0)$	(v_0, v_1, v_2, v_3)	$\emptyset$

Numbering of mesh entities on hexahedral cells

Entity	Incident vertices	Non-incident vertices
$v_0 = (0,0)$	(v_0)	$(v_1, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_1 = (0,1)$	(v_1)	$(v_0, v_2, v_3, v_4, v_5, v_6, v_7)$
$v_2 = (0,2)$	(v_2)	$(v_0, v_1, v_3, v_4, v_5, v_6, v_7)$
$v_3 = (0,3)$	(v_3)	$(v_0, v_1, v_2, v_4, v_5, v_6, v_7)$
$v_4 = (0,4)$	(v_4)	$(v_0, v_1, v_2, v_3, v_5, v_6, v_7)$
$v_5 = (0,5)$	(v_5)	$(v_0, v_1, v_2, v_3, v_4, v_6, v_7)$
$v_6 = (0,6)$	(v_6)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_7)$
$v_7 = (0,7)$	(v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6)$
$e_0 = (1,0)$	(v_6, v_7)	$(v_0, v_1, v_2, v_3, v_4, v_5)$
$e_1 = (1,1)$	(v_5, v_6)	$(v_0, v_1, v_2, v_3, v_4, v_7)$
$e_2 = (1,2)$	(v_4, v_7)	$(v_0, v_1, v_2, v_3, v_5, v_6)$
$e_3 = (1,3)$	(v_4, v_5)	$(v_0, v_1, v_2, v_3, v_6, v_7)$
$e_4 = (1,4)$	(v_3, v_7)	$(v_0, v_1, v_2, v_4, v_5, v_6)$
$e_5 = (1,5)$	(v_2, v_6)	$(v_0, v_1, v_3, v_4, v_5, v_7)$
$e_6 = (1,6)$	(v_2, v_3)	$(v_0, v_1, v_4, v_5, v_6, v_7)$
$e_7 = (1,7)$	(v_1, v_5)	$(v_0, v_2, v_3, v_4, v_6, v_7)$
$e_8 = (1,8)$	(v_1, v_2)	$(v_0, v_3, v_4, v_5, v_6, v_7)$
$e_9 = (1,9)$	(v_0, v_4)	$(v_1, v_2, v_3, v_5, v_6, v_7)$
$e_{10} = (1,10)$	(v_0, v_3)	$(v_1, v_2, v_4, v_5, v_6, v_7)$
$e_{11} = (1,11)$	(v_0, v_1)	$(v_2, v_3, v_4, v_5, v_6, v_7)$
$f_0 = (2,0)$	(v_4, v_5, v_6, v_7)	(v_0, v_1, v_2, v_3)
$f_1 = (2,1)$	(v_2, v_3, v_6, v_7)	(v_0, v_1, v_4, v_5)
$f_2 = (2,2)$	(v_1, v_2, v_5, v_6)	(v_0, v_3, v_4, v_7)
$f_3 = (2,3)$	(v_0, v_3, v_4, v_7)	(v_1, v_2, v_5, v_6)
$f_4 = (2,4)$	(v_0, v_1, v_4, v_5)	(v_2, v_3, v_6, v_7)
$f_5 = (2,5)$	(v_0, v_1, v_2, v_3)	(v_4, v_5, v_6, v_7)
$c_0 = (3,0)$	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$	$\emptyset$

1.5 Discussion

UFC has been used for many applications, including the Poisson equation; convection–diffusion–reaction equations; continuum equations for linear elasticity, hyperelasticity, and plasticity; the incompressible Navier–Stokes equations; mixed formulations for the Hodge Laplacian; and many more. The types of finite elements involved include standard continuous Lagrange elements of arbitrary order, discontinuous Galerkin formulations, Brezzi–Douglas–Marini elements, Raviart–Thomas elements, Crouzeix–Raviart elements, and Nedelec elements.

The form compilers FFC and SFC described in Chapters [logg-1] and [alnes-2] are UFC compliant, both generating efficient UFC code from an abstract problem definition. The assembler in DOLFIN uses the generated UFC code, communicates with the DOLFIN mesh data structure to extract `ufc::mesh` and `ufc::cell` data, and assembles the global tensor

into a data structure implemented by one of a number of linear algebra backends supported by DOLFIN, including PETSc, Trilinos (Epetra), uBLAS, and MTL4.

One of the main limitations in the current version (1.4) of the UFC interface is the assumption of a homogeneous mesh, that is, only one cell shape is allowed throughout the mesh. Thus, although mesh ordering conventions have been defined for the interval, triangle, tetrahedron, quadrilateral, and hexahedron, only one type of shape can be used at any time. Another limitation is that only one fixed finite element space can be chosen for each argument of the form, which excludes *p*-refinement (increasing the element order in a subset of the cells). These limitations may be addressed in future versions of the UFC interface.

1.6 Historical notes

UFC was introduced in 2007 when the first version of UFC (1.0) was released. The UFC interface has been used by DOLFIN since the release of DOLFIN 0.7.0 in 2007. The 1.0 release of UFC was followed by version 1.1 in 2008, version 1.2 in 2009, and version 1.4 in 2010. The new releases have involved minor corrections to the initial UFC interface but have also introduced some new functionality, like functions for evaluating multiple degrees of freedom (`evaluate_dofs` in addition to `evaluate_dof`) and multiple basis functions (`evaluate_basis_all` in addition to `evaluate_basis`). In contrast to other FEniCS components, few changes are made to the UFC interface in order to have a stable interface for both form compilers (FFC and SFC) and assemblers (DOLFIN).

Acknowledgment

The authors would like to thank Johan Hake, Ola Skavhaug, Garth Wells, Kristian Ølgaard, and Hans Petter Langtangen for their contributions to UFC.

Bibliography

[1] M. ALNÆS, H. P. LANGTANGEN, A.LOGG, K.-A. MARDAL, AND O. SKAVHAUG, *UFC Specification and User Manual*, 2007. URL: `http://www.fenics.org/ufc/`.

[2] M. S. ALNÆS, A. LOGG, K.-A. MARDAL, O. SKAVHAUG, AND H. P. LANGTANGEN, *Unified framework for finite element assembly*, International Journal of Computational Science and Engineering, 4 (2009), pp. 231–244.