UFC Specification and User Manual 1.1

August 18, 2009

Martin Sandve Alnæs, Anders Logg, Kent-Andre Mardal, Ola Skavhaug, and Hans Petter Langtangen

www.fenics.org

Visit `http://www.fenics.org/` for the latest version of this manual.
Send comments and suggestions to `ufc-dev@fenics.org`.

Contents

Chapter 1

Introduction

Large parts of a finite element program are similar from problem to problem and can therefore be coded as a general, reusable library. Mesh data structures, linear algebra and finite element assembly are examples of operations that are naturally coded in a problem-independent way and made available in reusable libraries [7, 3, 15, 4, 1, 5]. However, some parts of a finite element program are difficult to code in a problem-independent way. In particular, this includes the evaluation of the *element tensor* (the "element stiffness matrix"), that is, the evaluation of the local contribution from a finite element to a global sparse tensor (the "stiffness matrix") representing a discretized differential operator. These parts must thus be implemented by the application programmer for each specific combination of differential equation and discretization (finite element spaces).

However, domain-specific compilers such as FFC [14, 10, 13, 11, 12] and SyFi [2] make it possible to automatically generate the code for the evaluation of the element tensor. These *form compilers* accept as input a high-level description of a finite element variational form and generate low-level code code for efficient evaluation of the element tensor and associated quantities. It thus becomes important to specify the *interface* between form compilers and finite element assemblers such that the code generated by FFC, SyFi and other form compilers can be used to assemble finite element matrices and vectors (and in general tensors).

1.1 Unified Form-assembly Code

UFC (Unified Form-assembly Code) is a unified framework for finite element assembly. More precisely, it defines a fixed interface for communicating low level routines (functions) for evaluating and assembling finite element variational forms. The UFC interface consists of a single header file `ufc.h` that specifies a C++ interface that must be implemented by code that complies with the UFC specification.

Both FFC (since version 0.4.0) and SyFi (since version 0.3.4) generate code that complies with the UFC specification. Thus, code generated by FFC and SyFi may be used interchangeably by any UFC-based finite element assembler, such as DOLFIN [8].

1.2 Aim and scope

The UFC interface has been designed to make a minimal amount of assumptions on the form compilers generating the UFC code and the assemblers built on top of the UFC specification. Thus, the UFC specification provides a minimal amount of abstractions and data structures. Programmers wishing to implement the UFC specification will typically want to create system-specific (but simple) wrappers for the generated code.

Few assumptions have also been made on the underlying finite element methodology. The current specification is limited to affinely mapped cells, but does not restrict the mapping of finite element function spaces. Thus, UFC code may be generated for elements where basis functions are transformed from the reference cell by the affine mapping, as well as for elements where the basis functions must be transformed by the Piola mapping. UFC code has been successfully generated and used in finite element codes for standard continuous Galerkin methods (Lagrange finite elements of arbitrary order), discontinuous Galerkin methods (including integrals of jumps and averages over interior facets) and mixed methods (including Brezzi–Douglas–Marini and Raviart–Thomas elements).

1.3 Outline

In the next section, we give an overview of finite element assembly and explain how the code generated by form compilers may be used as the basic building blocks in the assembly algorithm. We then present the UFC interface in detail in Section 3. In Section 4 and Section 5, we define the reference cells and numbering conventions that must be followed by UFC-based form compilers and assemblers.

Chapter 2

Finite element assembly

In this section, we present a general algorithm for assembly of finite element variational forms and define the concepts that the UFC interface is based on.

2.1 Finite Element Discretization

2.1.1 The Finite Element

A finite element is mathematically defined as a triplet consisting of a polygon, a polynomial function space, and a set of linear functionals, see [6]. Given that the dimension of the function space and the number of the (linearly independent) linear functionals are equal, the finite element is uniquely defined. Hence, we will refer to a finite element as a collection of

- a polygon K,
- a polynomial space $\mathcal{P}_K$ on K,
- a set of linearly independent linear functionals, the *degrees of freedom*, $L_i : \mathcal{P}_K \to \mathbb{R}$, $i = 1, 2, \ldots, n$.

2.1.2 Variational Forms

Consider the weighted Poisson problem $-\nabla \cdot (w\nabla u) = f$ with Dirichlet boundary conditions on a domain $\Omega \subset \mathbb{R}^d$. Multiplying by a test function $v \in V_h$ and integrating by parts, one obtains the variational problem

$$\int_\Omega w\nabla v \cdot \nabla u \,\mathrm{d}x = \int_\Omega v f \,\mathrm{d}x \quad \forall v \in V_h, \tag{2.1}$$

for $u \in V_h$. If $w, f \in W_h$ for some discrete finite element space W_h (which may be different from V_h), we may thus write (2.1) as

$$a(v, u; w) = L(v; f) \quad \forall v \in V_h, \tag{2.2}$$

where the trilinear form $a : V_h \times V_h \times W_h \to \mathbb{R}$ is given by

$$a(v, u; w) = \int_\Omega w\nabla v \cdot \nabla u \,\mathrm{d}x \tag{2.3}$$

and the bilinear form $L : V_h \times W_h \to R$ is given by

$$L(v; f) = \int_\Omega v f \,\mathrm{d}x. \tag{2.4}$$

Note here that a is *bilinear* for any given fixed $w \in W_h$ and L is *linear* for any given fixed $f \in W_h$.

In general, we shall be concerned with the discretization of finite element variational forms of general arity $r + n > 0$,

$$a : V_h^1 \times V_h^2 \times \cdots \times V_h^r \times W_h^1 \times W_h^2 \times \cdots \times W_h^n \to \mathbb{R}, \tag{2.5}$$

defined on the product space $V_h^1 \times V_h^2 \times \cdots \times V_h^r \times W_h^1 \times W_h^2 \times \cdots \times W_h^n$ of two sets $\{V_h^j\}_{j=1}^r, \{W_h^j\}_{j=1}^n$ of discrete finite element function spaces on Ω. We refer to $(v_1, v_2, \ldots, v_r) \in V_h^1 \times V_h^2 \times \cdots \times V_h^r$ as *primary arguments*, and to $(w_1, w_2, \ldots, w_n) \in W_h^1 \times W_h^2 \times \cdots \times W_h^n$ as *coefficients* and write

$$a = a(v_1, \ldots, v_r; w_1, \ldots, w_n). \tag{2.6}$$

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.

2.1.3 Discretization

To discretize the form a, we introduce bases $\{\phi_i^1\}_{i=1}^{N^1}, \{\phi_i^2\}_{i=1}^{N^2}, \ldots, \{\phi_i^r\}_{i=1}^{N^r}$ for the function spaces $V_h^1, V_h^2, \ldots, V_h^r$ respectively and let $i = (i_1, i_2, \ldots, i_r)$ be a multiindex of length $|i| = r$. The form a then defines a rank r tensor given by

$$A_i = a(\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r; w_1, w_2, \ldots, w_n) \quad \forall i \in \mathcal{I}, \tag{2.7}$$

where $\mathcal{I}$ is the index set

$$\begin{aligned} \mathcal{I} &= \prod_{j=1}^{r} [1, |V_h^j|] = \\ &\{(1, 1, \ldots, 1), (1, 1, \ldots, 2), \ldots, (N^1, N^2, \ldots, N^r)\}. \end{aligned} \tag{2.8}$$

We refer to the tensor A as the *discrete operator* generated by the form a and the particular choice of basis functions. For any given form of arity $r + n$, the tensor A is a (typically sparse) tensor of rank r and dimension $|V_h^1| \times |V_h^2| \times \ldots \times |V_h^r| = N^1 \times N^2 \times \ldots \times N^r$.

Typically, the rank r is 0, 1, or 2. When $r = 0$, the tensor A is a scalar (a tensor of rank zero), when $r = 1$, the tensor A is a vector (the "load vector") and when $r = 2$, the tensor A is a matrix (the "stiffness matrix"). Forms of higher arity also appear, though they are rarely assembled as a higher-dimensional sparse tensor.

Note here that we consider the functions $w_1, w_2, \ldots, w_n$ as fixed in the sense that the discrete operator A is computed for a given set of functions, which we refer to as *coefficients*. As an example, consider again the variational problem (2.1) for the weighted Poisson's equation. For the trilinear form a, the rank is $r = 2$ and the number of coefficients is $n = 1$, while for the linear form L, the rank is $r = 1$ and the number of coefficients is $n = 1$. We may also choose to directly compute the *action* of the form a obtained by assembling a vector from the form

$$a(v_1; w_1, w2) = \int_\Omega w_1 \nabla v_1 \cdot \nabla w_2 \, \mathrm{d}x, \tag{2.9}$$

where now $r = 1$ and $n = 2$.

We list below a few other examples to illustrate the notation.

EXAMPLE **2.1.1** *Our first example is related to the divergence constraint in fluid flow. Let the form a be given by*

$$a(q, u) = \int_\Omega q \nabla \cdot u \, \mathrm{d}x, \quad q \in V_h^1, \quad u \in V_h^2, \tag{2.10}$$

where V_h^1 is a space of scalar-valued functions and where V_h^2 is a space of vector-valued functions. The form $a : V_h^1 \times V_h^2 \to \mathbb{R}$ has two primary arguments and thus $r = 2$. Furthermore, the form does not depend on any coefficients and thus $n = 0$.

EXAMPLE **2.1.2** *Another common form in fluid flow (with variable density) is*

$$a(v, u; w, \varrho) = \int_\Omega v \, \varrho \, w \cdot \nabla u \, \mathrm{d}x. \tag{2.11}$$

Here, $v \in V_h^1$, $u \in V_h^2$, $w \in W_h^1$, $\varrho \in W_h^2$, where V_h^1, V_h^2, and W_h^1 are spaces of vector-valued functions, while W_h^2 is a space of scalar-valued functions. The form takes four arguments, where two of the arguments are coefficients,

$$a : V_h^1 \times V_h^2 \times W_h^1 \times W_h^2 \to \mathbb{R}. \tag{2.12}$$

Hence, $r = 2$ and $n = 2$.

EXAMPLE **2.1.3** *The $H^1(\Omega)$ norm of the error $e = u - u_h$ squared is*

$$a(; u, u_h) = \int_\Omega (u - u_h)^2 + |\nabla (u - u_h)|^2 \, \mathrm{d}x. \tag{2.13}$$

The form takes two arguments and both are coefficients,

$$a : W_h^1 \times W_h^2 \to \mathbb{R}. \tag{2.14}$$

Hence, $r = 0$ and $n = 2$.

2.2 Finite Element Assembly

The standard algorithm for computing the global sparse tensor A is known as *assembly*, see [16, 9]. By this algorithm, the tensor A may be computed

by assembling (summing) the contributions from the local entities of a finite element mesh. To express this algorithm for assembly of the global sparse tensor A for a general finite element variational form of arity r, we introduce the following notation and assumptions.

Let $\mathcal{T} = \{K\}$ be a set of disjoint *cells* (a triangulation) partitioning the domain $\Omega = \cup_{K \in \mathcal{T}} K$. Further, let $\partial_e \mathcal{T}$ denote the set of *exterior facets* (the set of cell facets incident with the boundary $\partial\Omega$), and let $\partial_i \mathcal{T}$ denote the set of *interior facets* (the set of cell facets non-incident with the boundary $\partial\Omega$). For each discrete function space V_h^j, $j = 1, 2, \ldots, r$, we assume that the global basis $\{\phi_i^j\}_{i=1}^{N^j}$ is obtained by patching together local function spaces $\mathcal{P}_K^j$ on each cell K as determined by a local-to-global mapping.

We shall further assume that the variational form (2.5) may be expressed as a sum of integrals over the cells $\mathcal{T}$, the exterior facets $\partial_e \mathcal{T}$ and the interior facets $\partial_i \mathcal{T}$. We shall allow integrals 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.

We thus assume that the form a is given by

$$\begin{aligned} a(v_1, \ldots, v_r; w_1, \ldots, w_n) = & \\ & \sum_{k=1}^{n_c} \sum_{K \in \mathcal{T}_k} \int_K I_k^c(v_1, \ldots, v_r; w_1, \ldots w_n) \, \mathrm{d}x \\ & + \sum_{k=1}^{n_e} \sum_{S \in \partial_e \mathcal{T}_k} \int_S I_k^e(v_1, \ldots, v_r; w_1, \ldots, w_n) \, \mathrm{d}s \\ & + \sum_{k=1}^{n_i} \sum_{S \in \partial_i \mathcal{T}_k} \int_S I_k^i(v_1, \ldots, v_r; w_1, \ldots, w_n) \, \mathrm{d}s. \end{aligned} \tag{2.15}$$

We refer to an integral over a cell K 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 as an *interior facet integral* (typically used in discontinuous Galerkin methods).

For simplicity, we consider here initially assembly of the global sparse tensor A corresponding to a form a given by a single integral over all cells $\mathcal{T}$, and later extend to the general case where we must also account for contributions

from several cell integrals, interior facet integrals and exterior facet integrals.

We thus consider the form

$$\begin{aligned} &a(v_1, \ldots, v_r; w_1, \ldots, w_n) = \\ &\quad \sum_{K \in \mathcal{T}} \int_K I^c(v_1, \ldots, v_r; w_1, \ldots, w_n) \, \mathrm{d}x, \end{aligned} \tag{2.16}$$

for which the global sparse tensor A is given by

$$A_i = \sum_{K \in \mathcal{T}} \int_K I^c(\phi^1_{i_1}, \ldots, \phi^r_{i_r}; w_1, \ldots, w_n) \, \mathrm{d}x. \tag{2.17}$$

To see how to compute the tensor A by summing the local contributions from each cell K, we let $n^j_K = |\mathcal{P}^j_K|$ denote the dimension of the local finite element space on K for the jth primary argument $v_j \in V^j_h$ for $j = 1, 2, \ldots, r$. Furthermore, let

$$\iota^j_K : [1, n^j_K] \to [1, N^j] \tag{2.18}$$

denote the local-to-global mapping for V^j_h, that is, on any given $K \in \mathcal{T}$, the mapping ι^j_K maps the number of a local degree of freedom (or, equivalently, local basis function) to the number of the corresponding global degree of freedom (or, equivalently, global basis function). We then define for each $K \in \mathcal{T}$ the collective local-to-global mapping $\iota_K : \mathcal{I}_K \to \mathcal{I}$ by

$$\iota_K(i) = (\iota^1_K(i_1), \iota^2_K(i_2), \ldots, \iota^r_K(i_r)) \quad \forall i \in \mathcal{I}_K, \tag{2.19}$$

where $\mathcal{I}_K$ is the index set

$$\begin{aligned} \mathcal{I}_K &= \prod_{j=1}^{r} [1, |\mathcal{P}^j_K|] \\ &= \{(1, 1, \ldots, 1), (1, 1, \ldots, 2), \ldots, (n^1_K, n^2_K, \ldots, n^r_K)\}. \end{aligned} \tag{2.20}$$

Furthermore, for each V^j_h we let $\{\phi^{K,j}_i\}_{i=1}^{n^j_K}$ denote the restriction to an element K of the subset of the basis $\{\phi^j_i\}_{i=1}^{N^j} \subset \mathcal{P}^j_K$ of V^j_h supported on K.

We may now compute A by summing the contributions from the local cells,

$$\begin{aligned} A_i &= \sum_{K \in \mathcal{T}_i} \int_K I^c(\phi_{i_1}^1, \ldots, \phi_{i_r}^r; w_1, \ldots, w_n) \, \mathrm{d}x \\ &= \sum_{K \in \mathcal{T}_i} \int_K I^c(\phi_{(\iota_K^1)^{-1}(i_1)}^{K,1}, \ldots, \phi_{(\iota_K^r)^{-1}(i_r)}^{K,r}; w_1, \ldots, w_n) \, \mathrm{d}x \\ &= \sum_{K \in \mathcal{T}_i} A^K_{\iota_K^{-1}(i)}, \end{aligned} \tag{2.21}$$

where A^K is the local *cell tensor* on cell K (the "element stiffness matrix"), given by

$$A_i^K = \int_K I^c(\phi_{i_1}^{K,1}, \ldots, \phi_{i_r}^{K,r}; w_1, \ldots, w_n) \, \mathrm{d}x, \tag{2.22}$$

and where $\mathcal{T}_i$ denotes the set of cells on which all basis functions $\phi_{i_1}^1, \phi_{i_2}^2, \ldots, \phi_{i_r}^r$ are supported. Similarly, we may sum the local contributions from the exterior and interior facets in the form of local *exterior facet tensors* and *interior facet tensors*.

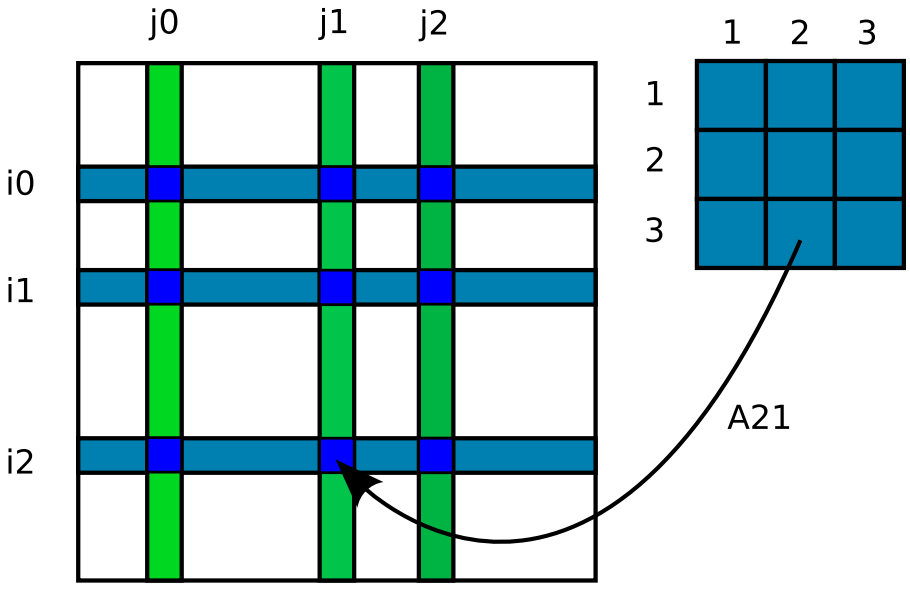

Figure 2.1: Adding the entries of a cell tensor A^K to the global tensor A using the local-to-global mapping ι_K, illustrated here for a rank two tensor (a matrix).

In Algorithm 1, we present a general algorithm for assembling the contributions from the local cell, exterior facet and interior facet tensors into a global sparse tensor. In all cases, we iterate over all entities (cells, exterior or interior facets), compute the local cell tensor A^K (or exterior/interior facet tensor A^S) and add it to the global sparse tensor as determined by the local-to-global mapping, see Figure 2.1.

Algorithm 1 Assembling the global tensor A from the local contributions on all cells, exterior and interior facets. For assembly over exterior facets, $K(S)$ refers to the cell $K \in \mathcal{T}$ incident to the exterior facet S, and for assembly over interior facets, $K(S)$ refers to the "macro cell" consisting of the pair of cells K^+ and K^- incident to the interior facet S.

$A = 0$
(i) *Assemble contributions from all cells*
for each $K \in \mathcal{T}$

for $j = 1, 2, \ldots, r$:
Tabulate the local-to-global mapping ι_K^j

for $j = 1, 2, \ldots, n$:
Extract the values of w_j on K

Take $0 \leq k \leq n_c$ such that $K \in \mathcal{T}_k$
Tabulate the cell tensor A^K for I_k^c
Add A_i^K to $A_{\iota_K^1(i_1), \iota_K^2(i_2), \ldots, \iota_K^r(i_r)}$ for $i \in I_K$

(ii) *Assemble contributions from all exterior facets*
for each $S \in \partial_e \mathcal{T}$

for $j = 1, 2, \ldots, r$:
Tabulate the local-to-global mapping $\iota_{K(S)}^j$

for $j = 1, 2, \ldots, n$:
Extract the values of w_j on $K(S)$

Take $0 \leq k \leq n_e$ such that $S \in \partial_e \mathcal{T}_k$
Tabulate the exterior facet tensor A^S for I_k^e
Add A_i^S to $A_{\iota_{K(S)}^1(i_1), \iota_{K(S)}^2(i_2), \ldots, \iota_{K(S)}^r(i_r)}$ for $i \in I_{K(S)}$

(iii) *Assemble contributions from all interior facets*
for each $S \in \partial_i \mathcal{T}$

for $j = 1, 2, \ldots, r$:
Tabulate the local-to-global mapping $\iota_{K(S)}^j$

for $j = 1, 2, \ldots, n$:
Extract the values of w_j on $K(S)$

Take $0 \leq k \leq n_i$ such that $S \in \partial_i \mathcal{T}_k$
Tabulate the interior facet tensor A^S for I_k^i
Add A_i^S to $A_{\iota_{K(S)}^1(i_1), \iota_{K(S)}^2(i_2), \ldots, \iota_{K(S)}^r(i_r)}$ for $i \in I_{K(S)}$

Chapter 3

Interface specification

3.1 A short remark on design

UFC is organized as a minimalistic set of abstract C++ classes representing low-level abstractions of the finite element method. The functions in these classes are mainly of two types: (i) functions returning dimensions, which are typically used to allocate an array, and (ii) functions that fill an array with values.

It is considered the assembly routine's responsibility to allocate and deallocate arrays of proper size. Consider for example the function for evaluating the ith basis function in the class `ufc::finite_element`:

```
virtual void evaluate_basis(unsigned int i,
                            double* values,
                            const double* coordinates,
                            const cell& c) const = 0;
```

This function assumes that the array `values` has the correct size, which may be obtained by calling the functions `value_rank` and `value_dimension` as described in detail below.

Thus, the UFC interface is a low-level interface that should be simple to integrate into an existing C++ finite element code, but which is probably not suitable to be exposed as part of an end-user interface.

The UFC interface is defined by a single header file `ufc.h` which defines the central interface class `ufc::form` and a small set of auxiliary interface classes. In addition, a pair of data structures `ufc::mesh` and `ufc::cell` are defined and used for passing data to the interface functions. All functions defined by the UFC interface are *pure virtual*, meaning that all these functions must be overloaded in each implementation of the classes. All but two functions (`init_mesh` and `init_cell`) are `const`, meaning that calling these `const` functions will leave the UFC objects unchanged. Input argument (pointers) are always `const`, while output arguments (pointers) are always non-`const`.

The interface is presented below in the same order as it is defined in the header file `ufc.h`. Thus, the interface is presented bottom-up, starting with the definition of basic data structures and ending with the definition of the main `ufc::form` interface class.

3.2 Cell shapes

```
enum shape {interval,
            triangle,
            quadrilateral,
            tetrahedron,
            hexahedron};
```

This enumeration includes all cell shapes that are covered by the UFC specification, see Chapter 4.

3.3 The class `ufc::mesh`

The class `ufc::mesh` defines a data structure containing basic information about an unstructured mesh. It is used for passing a minimal amount of information about the global mesh to UFC functions.

3.3.1 The integer `topological_dimension`

```
unsigned int topological_dimension;
```

The unsigned integer `topological_dimension` holds the topological dimension of the mesh, that is, the topological dimension of the cells of the mesh. For the supported cell shapes defined above, the topological dimensions are as follows: `interval` has dimension one, `triangle` and `quadrilateral` have dimension two, and `tetrahedron` and `hexahedron` have dimension three.

3.3.2 The integer `geometric_dimension`

```
unsigned int geometric_dimension;
```

The unsigned integer `geometric_dimension` holds the geometric dimension of the mesh, that is, the dimension of the coordinates of the mesh vertices. Often, the geometric dimension is equal to the topological dimension, but they may differ. For example, one may have a topologically two-dimensional mesh embedded in three-dimensional space.

3.3.3 The array `num_entities`

```
unsigned int* num_entities;
```

The array `num_entities` should contain the number of entities within each topological dimension of the mesh (see Chapter 4). The size of the array should be equal to the topological dimension of the mesh plus one.

Thus, for a mesh of tetrahedral cells, `num_entities[0]` should contain the number of vertices, `num_entities[1]` should contain the number of edges (if they are needed, see `ufc::dof_map::needs_mesh_entities` below), `num_entities[2]` should contain the number of faces, and `num_entities[3]` should contain the number of volumes. If `d` is the topological dimension of the mesh, `num_entities[d]` should contain the number of cells or elements.

3.4 The class `ufc::cell`

The class `ufc::cell` defines the data structure for a cell in a mesh. Its intended use is not as a building block in a mesh data structure, but merely as a view of specific data for a single cell. It is used to pass cell data to UFC functions with a minimal amount of assumptions on how the computational mesh is represented and stored.

3.4.1 The enum variable `cell_shape`

```
shape cell_shape;
```

The variable `cell_shape` should be set to the corresponding `ufc::shape` for the cell.

3.4.2 The integer `topological_dimension`

```
unsigned int topological_dimension;
```

The integer `topological_dimension` should be set to the topological dimension of the cell (see `ufc::mesh` above).

3.4.3 The integer `geometric_dimension`

```
unsigned int geometric_dimension;
```

The integer `geometric_dimension` should be set to the geometric dimension of the cell (see `ufc::mesh` above).

3.4.4 The array `entity_indices`

```
unsigned int** entity_indices;
```

The array `entity_indices` should contain the global indices for all entities of the cell (see Chapter 4). The length of the array `entity_indices` should be equal to the value of the variable `topological_dimension` plus one.

Thus, `entity_indices[0]` should be an array containing the global indices of all the vertices of the cell, `entity_indices[1]` should be an array containing the global indices of all the edges of the cell, etc. The sizes of these arrays are implicitly defined by the cell type.

Note that the entity indices are not always needed for all entities of the cell. Which entities are required is specified by the `ufc::dof_map` class (see `ufc::dof_map::needs_mesh_entities` below).

3.4.5 The array `coordinates`

```
double** coordinates;
```

The array `coordinates` should contain the global coordinates for all vertices of the cell and thus its length should be equal to number of vertices of the cell. The length of the array `coordinates[0]` should be equal to the value of the variable `geometric_dimension` and it should contain the x, y, ... coordinates of the first vertex etc.

3.5 The class `ufc::function`

The class `ufc::function` is an interface for evaluation of general tensor-valued functions on the cells of a mesh.

3.5.1 The function `evaluate`

```
virtual void evaluate(double* values,
                      const double* coordinates,
                      const cell& c) const = 0;
```

The only function in this class is `evaluate`, which evaluates all the value components of the function at a given point in a given cell of the mesh.

The output of `evaluate` should be written to the array `values`. For a scalar-valued function, a single value should be written to `values[0]`. For general tensor-valued functions, the values should be written in a flattened row-major ordering of the tensor values. Thus, for a function $f : K \to \mathbb{R}^{2x2}$ (where $A = f(x)$ is a 2×2 matrix), the array `values` should contain the values $A_{11}, A_{12}, A_{21}, A_{22}$ in this order.

The input to `evaluate` are the coordinates of a point in a cell and the UFC view of the cell containing that point.

See also the description of `ufc::finite_element::evaluate_dof` below.

3.6 The class `ufc::finite_element`

The class `ufc::finite_element` represents a finite element in the classical Ciarlet sense [6], or rather a particular instance of a finite element for a particular choice of nodal basis functions. Thus, a `ufc::finite_element` has functions for accessing the shape of the finite element, the dimension of the polynomial function space, the basis functions of the function space (and their derivatives), and the linear functionals defining the degrees of freedom. In addition, a `ufc::finite_element` provides functionality for interpolation.

3.6.1 The function `signature`

```
virtual const char* signature() const = 0;
```

This function returns a signature string that uniquely identifies the finite element. This can be used to compare whether or not two given `ufc::finite_element` objects are identical.

3.6.2 The function `cell_shape`

```
virtual shape cell_shape() const = 0;
```

This function returns the shape of the cell the finite element is defined on.

3.6.3 The function `space_dimension`

```
virtual unsigned int space_dimension() const = 0;
```

This function returns the dimension of the local finite element space ($|V_h^K|$), which is equal to the number of basis functions. This should also be equal to the value of `local_dimension()` for the corresponding `ufc::dof_map` (see below).

3.6.4 The function `value_rank`

```
virtual unsigned int value_rank() const = 0;
```

A finite element can have general tensor-valued basis functions. The function `value_rank` returns the rank of the value space of the basis functions. For a scalar element, this function should return zero, for vector-valued functions it should return one, for matrix-valued functions it should return two, etc. For mixed elements, this may not always make sense, for example with a tensor-vector-scalar element. Thus the value rank of a mixed element must be 1 if any of the subelements have different value ranks.

3.6.5 The function `value_dimension`

```
virtual unsigned int
value_dimension(unsigned int i) const = 0;
```

This function returns the dimension of the value space of the finite element basis functions for the given axis, where the given axis must be a number between zero and the value rank minus one.

Note that the total size (number of values) of the value space is obtained as the product of `value_dimension(i)` for $0 \leq i <$ `value_rank()`. For a mixed element with value rank 1 Since the value rank of a mixed element must be 1 if any of the subelements have different value ranks, `value_dimension(0)` is then the total value size of all the subelements.

3.6.6 The function `evaluate_basis`

```
virtual void evaluate_basis(unsigned int i,
                            double* values,
                            const double* coordinates,
                            const cell& c) const = 0;
```

This function evaluates basis function `i` at the given `coordinates` within the given cell `c`, and stores the values in the array `values`. The size of the output array should be equal to size of the value space (see `value_dimension` above).

3.6.7 The function `evaluate_basis_all`

Introduced in version 1.1.

```
virtual void evaluate_basis_all(double* values,
                                const double* coordinates,
                                const cell& c) const = 0;
```

As `evaluate_basis`, but evaluates all basis functions at once, which can be implemented much more effectively than multiple calls to `evaluate_basis`. The size of the output array should be equal to size of the value space times the number of basis functions. The computed values for a single basis function are placed contiguously in the array.

3.6.8 The function `evaluate_basis_derivatives`

```
virtual void
evaluate_basis_derivatives(unsigned int i,
                           unsigned int n,
```

```
                                double* values,
                                const double* coordinates,
                                const cell& c) const = 0;
```

This function evaluates all order `n` derivatives of basis function `i` at the given `coordinates` within the given `cell`, and stores the values in the array `values`. Derivatives may be obtained up to the polynomial degree of the finite element function space with higher degree derivatives evaluating to zero.

The number of derivatives is given by d^n where d is the geometric dimension of the cell. For $n = 1$, $d = 3$, the order of the derivatives is naturally $\partial/\partial x$, $\partial/\partial y$, $\partial/\partial z$. For $n = 2$, $d = 3$, the order of the derivatives is $\frac{\partial^2}{\partial x \partial x}, \frac{\partial^2}{\partial x \partial y}, \ldots, \frac{\partial^2}{\partial z \partial z}$. Thus, the derivatives are stored in a flattened row-major ordering based on the derivative spatial dimensions.

For tensor-valued basis functions, the ordering of derivatives is row-major based on the value space dimensions followed by the derivative spatial dimensions.

3.6.9 The function `evaluate_basis_derivatives_all`

Introduced in version 1.1.

```
virtual void
evaluate_basis_derivatives_all(unsigned int n,
                               double* values,
                               const double* coordinates,
                               const cell& c) const = 0;
```

As `evaluate_basis_derivatives`, but evaluated for all basis functions at once, which can be implemented much more effectively than multiple calls to `evaluate_basis_derivatives`. The size of the output array should be equal to the corresponding size defined for `evaluate_basis_derivatives`

times the number of basis functions. The computed values for a single basis function are placed contiguously in the array.

3.6.10 The function `evaluate_dof`

```
virtual double evaluate_dof(unsigned int i,
                            const function& f,
                            const cell& c) const = 0;
```

This function evaluates and returns the value of the degree of freedom `i` (which is a linear functional) on the given function `f`.

For example, the degrees of freedom for Lagrange finite elements are given by evaluation of `f` at a set of points. Other examples of degrees of freedom include weighted integrals over facets or normal components on facets.

3.6.11 The function `evaluate_dofs`

Introduced in version 1.1.

```
virtual void evaluate_dofs(double* values,
                           const function& f,
                           const cell& c) const;
```

Vectorized version of `evaluate_dof` for efficiency, returning the values of all degrees of freedom in the array `values`.

3.6.12 The function `interpolate_vertex_values`

```
virtual void
interpolate_vertex_values(double* vertex_values,
```

```
                                const double* dof_values,
                                const cell& c) const = 0;
```

This function takes as input the array `dof_values` containing the expansion coefficients for some function in the nodal basis and computes the values of that function at the vertices of the given cell, storing those values in the array `vertex_values`. If the function is tensor-valued, the values are stored in the array `vertex_values` row-major on the list of vertices followed by the row-major ordering of the tensor values as described above.

3.6.13 The function `num_sub_elements`

```
virtual unsigned int num_sub_elements() const = 0;
```

This function returns the number of subelements for a nested (mixed) element. For simple elements (non-nested), this function should return one.

A nested element is an element that is defined from a set of elements by taking the direct sum (tensor product) of the polynomial spaces of those elements. For example, the basis functions $\psi_1, \psi_2, \ldots, \psi_m$ of a vector-valued Lagrange element may be constructed from a scalar Lagrange element by repeating the basis functions $\phi_1, \phi_2, \ldots, \phi_n$ of the scalar element and padding with zeros: $\psi_1 = (\phi_1, 0), \psi_2 = (\phi_2, 0), \ldots, \psi_n = (\phi_n, 0), \psi_{n+1} = (0, \phi_1), \psi_{n+2} = (0, \phi_2), \ldots$.

Finite elements may be nested at arbitrary depth. For example, a mixed Taylor–Hood element may be created by combining a vector-valued quadratic Lagrange finite element with a scalar linear Lagrange finite element, and the vector-valued element may in turn be created by combining a set of scalar quadratic Lagrange elements.

3.6.14 The function `create_sub_element`

```
virtual finite_element*
create_sub_element(unsigned int i) const = 0;
```

This factory function constructs a `ufc::finite_element` object for subelement `i`. The argument `i` must be an integer between zero and the number of subelements (`num_sub_elements`) minus one. If the element is simple (non-nested), this function creates a copy of the finite element itself. The caller is responsible for deleting the returned object.

3.7 The class `ufc::dof_map`

This class represents the local-to-global mapping of degrees of freedom (dofs), or rather one particular instance of such a mapping (there are many possible local-to-global mappings) as defined in Equation (2.18). The most central function of this class is `tabulate_dofs`, which tabulates the local-to-global mapping from the degree of freedom indices on a local cell to a global vector of degree of freedom indices.

3.7.1 The function `signature`

```
virtual const char* signature() const = 0;
```

This function returns a signature string that uniquely identifies the dofmap. This can be used to compare whether or not two given `ufc::dof_map` objects are identical. (This may be used to optimize the assembly of forms by caching previously computed dofmaps.)

3.7.2 The function `needs_mesh_entities`

```
virtual bool needs_mesh_entities(unsigned int d) const = 0;
```

This function returns true if the `ufc::dof_map` requires mesh entities of topological dimension `d` to be available in `ufc::cell` arguments. Thus, if this function returns false for a given value of `d`, then the array `entity_indices[d]` of the `ufc::cell` data structure will not be used during calls to `ufc::dof_map` functions. In those cases, the array `entity_indices[d]` may be set to zero.

This may be used to check which entities must be generated to tabulate the local-to-global mapping. For example, linear Lagrange elements will only need to know the vertices of each cell in the mesh, while quadratic Lagrange elements will also need to know the edges of each cell in the mesh.

3.7.3 The function `init_mesh`

```
virtual bool init_mesh(const mesh& mesh) = 0;
```

This function initializes the dofmap for a given mesh. If it returns true, calls to `init_cell` and `init_cell_finalize` are required to complete the initialization. The function `global_dimension` be may only be called when the initialization is complete.

3.7.4 The function `init_cell`

```
virtual void init_cell(const mesh& m,
                       const cell& c) = 0;
```

For `ufc::dof_map` objects where `init_mesh` returns true, this function must be called for each cell in the mesh to initialize the dofmap.

3.7.5 The function `init_cell_finalize`

```
virtual void init_cell_finalize() = 0;
```

For `ufc::dof_map` objects where `init_mesh` returns true, this function must be called after `init_cell` is called for each cell in the mesh to complete initialization of the dofmap.

3.7.6 The function `global_dimension`

```
virtual unsigned int global_dimension() const = 0;
```

This function returns the dimension of the global finite element space on the mesh that the `ufc::dof_map` has been initialized for. The result of calling this function before the initialization is complete is undefined.

3.7.7 The function `local_dimension`

Changed in version 1.2.

```
virtual unsigned int local_dimension(const cell& c) const = 0;
```

This function returns the dimension of the local finite element space on a given cell.

3.7.8 The function `max_local_dimension`

Introduced in version 1.2.

```
virtual unsigned int max_local_dimension() const = 0;
```

This function returns the maximum dimension of the local finite element space on a single cell.

3.7.9 The function `geometric_dimension`

Introduced in version 1.1.

```
virtual unsigned int geometric_dimension() const;
```

This function returns the geometric dimension of the coordinates returned by `tabulate_coordinates`.

3.7.10 The function `num_facet_dofs`

```
virtual unsigned int num_facet_dofs() const = 0;
```

This function returns the number of dofs associated with a single facet of a cell, including all dofs associated with mesh entities of lower dimension incident with the facet. For example on a tetrahedron this will include dofs associated with edges and vertices of the triangle face. This is also the number of dofs that should be set if a Dirichlet boundary condition is applied to a single facet.

3.7.11 The function `num_entity_dofs`

Introduced in version 1.1.

```
virtual unsigned int num_entity_dofs(unsigned int d) const;
```

This function returns the number of dofs associated with a single mesh entity of dimension `d` in a cell, not including dofs associated with incident entities of lower dimension (unlike `num_facet_dofs()`). It is assumed that all cells of the mesh have the same number of degrees of freedom on each mesh entity of the same dimension.

3.7.12 The function `tabulate_dofs`

```
virtual void tabulate_dofs(unsigned int* dofs,
                           const mesh& m,
                           const cell& c) const = 0;
```

This function tabulates the global dof indices corresponding to each dof on the given cell. The size of the output array `dofs` should be equal to the value returned by `local_dimension()`.

3.7.13 The function `tabulate_facet_dofs`

```
virtual void
tabulate_facet_dofs(unsigned int* dofs,
                    unsigned int facet) const = 0;
```

This function tabulates the local dof indices associated with a given local facet, including all dofs associated with mesh entities of lower dimension incident with the facet. The size of the output array `dofs` should equal the value returned by `num_facet_dofs`.

3.7.14 The function `tabulate_entity_dofs`

Introduced in version 1.1.

```
virtual void tabulate_entity_dofs(unsigned int* dofs,
                                  unsigned int d,
                                  unsigned int i) const;
```

This function tabulates the local dof indices associated with a given local mesh entity `i` of dimension `d`, i.e. mesh entity (d, i), not including dofs associated with incident entities of lower dimension (unlike `tabulate_facet_dofs`). The size of the output array `dofs` should equal the value returned by the function `num_entity_dofs(d)`.

As an example, calling `tabulate_entity_dofs` for a face (d = 2) should yield only the dofs associated with the face that are not associated with vertices and edges. Thus `tabulate_entity_dofs` can be used to build connectivity information.

3.7.15 The function `tabulate_coordinates`

```
virtual void tabulate_coordinates(double** coordinates,
                                  const cell& c) const = 0;
```

This function tabulates the coordinates for each dof on the given cell. For Lagrange elements, this function will tabulate a set of points on the given cell such that the dofs of the finite element are given by evaluation at those points.

For elements that do not have a direct relationship between coordinates and dofs, an attempt should be made at a sensible implementation of this function. For example, if a dof is defined as the integral over a facet, the midpoint of the facet can be used. If no other choice makes sense, the midpoint of the

cell can be used as a last resort. This function must thus be used with care if non-Lagrangian elements are used.

The size of the output array `coordinates` should be equal to the value returned by `local_dimension()` and the size of each subarray `coordinates[0]`, `coordinates[1]` etc should be equal to the geometric dimension of the mesh, which can be obtained with the function `dof_map::geometric_dimension()`.

3.7.16 The function `num_sub_dof_maps`

```
virtual unsigned int num_sub_dof_maps() const = 0;
```

This function returns the number of sub-dofmaps for a nested (mixed) element. For a discussion on the subelement concept, see the documentation of the function `ufc::finite_element::num_sub_elements`. For simple elements (non-nested), this function should return one.

3.7.17 The function `create_sub_dof_map`

```
virtual dof_map* create_sub_dof_map(unsigned int i) const = 0;
```

This factory function constructs a `ufc::dof_map` object for subelement `i`. The argument `i` must be a number between zero and the number of sub-dofmaps (`num_sub_dof_maps`) minus one. If the dofmap is simple (non-nested), this function creates a copy of the dofmap itself. The caller is responsible for deleting the returned object.

3.8 The integral classes

As described in Section 2, and in particular Equation (2.15), the global sparse tensor (the "stiffness matrix") representing a given form (differential oper-

ator) may be assembled by summing the contributions from the local cells, exterior facets and interior facets of the mesh.

These contributions are represented in the UFC interface by the classes `cell_integral`, `exterior_facet_integral` and `interior_facet_integral`. Each of these three integral classes has a single function `tabulate_tensor` which may be called to compute the corresponding local contribution (cell tensor, exterior facet tensor or interior facet tensor).

3.9 The class `ufc::cell_integral`

The class `ufc::cell_integral` represents the integral of a form over a local cell in a finite element mesh. It has a single function `tabulate_tensor` which may be called to tabulate the values of the cell tensor for a given cell.

3.9.1 The function `tabulate_tensor`

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const cell& c) const = 0;
```

This function tabulates the values of the cell tensor for a form into the given array `A`. The size of this array should be equal to the product of the local space dimensions for the set of finite element function spaces corresponding to the arguments of the form. For example, when computing the matrix for a bilinear form defined on piecewise linear scalar elements on triangles, the space dimension of the local finite element is three and so the size of the array `A` should be $3 \times 3 = 9$.

The array `w` should contain the expansion coefficients for all *coefficients* of the form in the finite element nodal basis for each corresponding function space. Thus, the size of the array `w` should be equal to the number of coefficients n, and the size of each each array `w[0]`, `w[1]` etc should be equal to the space dimension of the corresponding local finite element space.

3.10 The class `ufc::exterior_facet_integral`

The class `ufc::exterior_facet_integral` represents the integral of a form over a local exterior facet (boundary facet) in a finite element mesh. It has a single function `tabulate_tensor` which may be called to tabulate the values of the exterior facet tensor for a given facet.

3.10.1 The function `tabulate_tensor`

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const cell& c,
                             unsigned int facet) const = 0;
```

The arrays `A` and `w` have the same function and should have the same sizes as described in the documentation for `cell_integral::tabulate_tensor`. Thus, the values of the exterior facet integral will be tabulated into the array `A` and the nodal basis expansions of all coefficients should be provided in the array `w`.

The additional argument `facet` should specify the local number of the facet with respect to its (single) incident cell. Thus, when the facet is an edge of a triangle, the argument `facet` should be an integer between zero and two (0, 1, 2) and when the facet is a facet of a tetrahedron, the argument `facet` should be an integer between zero and three (0, 1, 2, 3).

3.11 The class `ufc::interior_facet_integral`

The class `ufc::interior_facet_integral` represents the integral of a form over a local interior facet in a finite element mesh. It has a single function `tabulate_tensor` which may be called to tabulate the values of the interior facet tensor for a given facet.

3.11.1 The function `tabulate_tensor`

```
virtual void tabulate_tensor(double* A,
                             const double * const * w,
                             const cell& c0,
                             const cell& c1,
                             unsigned int facet0,
                             unsigned int facet1) const = 0;
```

Just as for the `cell_integral` and `exterior_facet_integral` classes, the `tabulate_tensor` function for the class `interior_facet_integral` tabulates the values of the local (interior facet) tensor into the array `A`, given the nodal basis expansions of the form coefficients in the array `w`. However, the interior facet tensor contains contributions from the two incident cells of an interior facet and thus the dimensions of these arrays are different.

On each interior facet, the two incident (neighboring) cells form a "macro cell" consisting of the total set of local basis functions on the two cells. The set of basis functions on the macro element is obtained by extending the basis functions on each of the two cells by zero to the macro cell. Thus, the space dimension of the finite element function space on the macro element is twice the size of the finite element function space on a single cell. The ordering of basis functions and degrees of freedom on the macro cell is obtained by first enumerating the basis functions and degrees of freedom on one of the two cells and then the basis functions and degrees of freedom on the second cell.

Thus the size of the array `A` should be equal to the product of twice the local space dimensions for the set of finite element function spaces corresponding to the arguments of the form. For example, when computing the matrix for a bilinear form defined on piecewise linear elements on triangles, the space dimension of the local finite element is three and so the size of the array `A` should be $6 \times 6 = 36$.

Similarly, the array `w` should contain the expansion coefficients for all *coefficients* of the form in the finite element nodal basis for each corresponding function space on the macro cell. Thus, the size of the array `w` should be equal to the number of coefficients n and the size of each each array `w[0]`,

`w[1]` etc should be equal to twice the space dimension of the corresponding local finite element space.

The additional arguments `facet0` and `facet1` should specify the local number of the facet with respect to its two incident cells. Thus, when the facet is an edge of a triangle, each of these arguments may be an integer between zero and two (0, 1, 2) and when the facet is a face of a tetrahedron, each of these arguments may be an integer between zero and three (0, 1, 2, 3).

3.12 The class `ufc::form`

The `ufc::form` class is the central part of the UFC interface and it represents a form

$$a = a(v_1, \ldots, v_r; w_1, \ldots, w_n), \tag{3.1}$$

defined on the product space $V_h^1 \times V_h^2 \times \cdots \times V_h^r \times W_h^1 \times W_h^2 \times \cdots \times W_h^n$ of two sets $\{V_h^j\}_{j=1}^r, \{W_h^j\}_{j=1}^n$ of finite element function spaces on a triangulation $\mathcal{T}$ of a domain $\Omega \subset \mathbb{R}^d$.

A `ufc::form` provides functions for accessing the rank r and number of coefficients n for a form, and factory functions for creating UFC objects for the corresponding cell integrals, exterior facet integrals, interior facet integrals, and all associated finite elements and dofmaps (local-to-global mappings).

3.12.1 The function `signature`

```
virtual const char* signature() const = 0;
```

This function returns a signature string that uniquely identifies the form. This can be used to compare whether or not two given `ufc::form` objects are identical.

3.12.2 The function `rank`

```
virtual unsigned int rank() const = 0;
```

This function returns the rank r of the global tensor generated by the form (the arity of the form).

3.12.3 The function `num_coefficients`

```
virtual unsigned int num_coefficients() const = 0;
```

This function returns the number of coefficients n for the form. Note that all integral terms of a form must have the same coefficients, even if not all coefficients are present in each term of the form.

3.12.4 The function `num_cell_integrals`

```
virtual unsigned int num_cell_integrals() const = 0;
```

This function returns the number of different cell integrals for the form. A form may have an arbitrary number of integrals over disjoint subdomains of the mesh.

3.12.5 The function `num_exterior_facet_integrals`

```
virtual unsigned int num_exterior_facet_integrals() const = 0;
```

This function returns the number of different exterior facet integrals for the form. A form may have an arbitrary number of integrals over disjoint subdomains of the mesh boundary.

3.12.6 The function `num_interior_facet_integrals`

```
virtual unsigned int num_interior_facet_integrals() const = 0;
```

This function returns the number of different interior facet integrals for the form. A form may have an arbitrary number of integrals over disjoint subsets of the interior facets of the mesh.

3.12.7 The function `create_finite_element`

```
virtual finite_element*
create_finite_element(unsigned int i) const = 0;
```

This factory function constructs a `ufc::finite_element` object for form argument `i`. A form with rank r and number of coefficients n has $r + n$ arguments, so this function returns the finite element object for tensor axis i if $i < r$, or the finite element for coefficient $i - r$ if $i \geq r$. The caller is responsible for deleting the returned object.

3.12.8 The function `create_dof_map`

```
virtual dof_map*
create_dof_map(unsigned int i) const = 0;
```

This factory function constructs a `ufc::dof_map` object for form argument `i`. A form with rank r and number of coefficients n has $r + n$ arguments, so this function returns the dofmap object for tensor axis i if $i < r$, or the dofmap for coefficient $i - r$ if $i \geq r$. The caller is responsible for deleting the returned object.

3.12.9 The function `create_cell_integral`

```
virtual cell_integral*
create_cell_integral(unsigned int i) const = 0;
```

This factory function constructs a `cell_integral` object for cell domain `i`. The caller is responsible for deleting the returned object.

3.12.10 The function `create_exterior_facet_integral`

```
virtual exterior_facet_integral*
create_exterior_facet_integral(unsigned int i) const = 0;
```

This factory function constructs an `exterior_facet_integral` object for exterior facet domain `i`. The caller is responsible for deleting the returned object.

3.12.11 The function `create_interior_facet_integral`

```
virtual interior_facet_integral*
create_interior_facet_integral(unsigned int i) const = 0;
```

This factory function constructs an `interior_facet_integral` object for interior facet domain `i`. The caller is responsible for deleting the returned object.

Chapter 4

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 4.1).

The UFC specification assumes that each cell in a finite element mesh is always isomorphic to one of the reference cells.

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 4.1: Reference cells covered by the UFC specification.

Figure 4.1: The reference interval.

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

Table 4.2: Vertex coordinates of the reference interval.

4.1 The reference interval

The reference interval is shown in Figure 4.1 and is defined by its two vertices with coordinates as specified in Table 4.2.

4.2 The reference triangle

The reference triangle is shown in Figure 4.2 and is defined by its three vertices with coordinates as specified in Table 4.3.

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

Table 4.3: Vertex coordinates of the reference triangle.

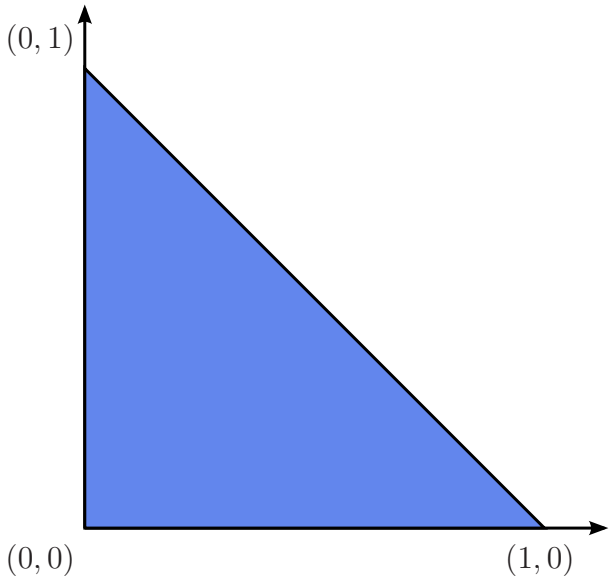

Figure 4.2: The reference triangle.

4.3 The reference quadrilateral

The reference quadrilateral is shown in Figure 4.3 and is defined by its four vertices with coordinates as specified in Table 4.4.

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

Table 4.4: Vertex coordinates of the reference quadrilateral.

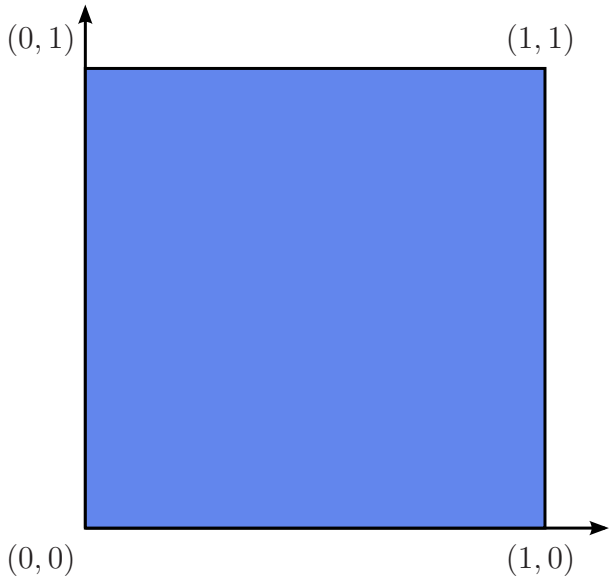

Figure 4.3: The reference quadrilateral.

4.4 The reference tetrahedron

The reference tetrahedron is shown in Figure 4.4 and is defined by its four vertices with coordinates as specified in Table 4.5.

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 4.5: Vertex coordinates of the reference tetrahedron.

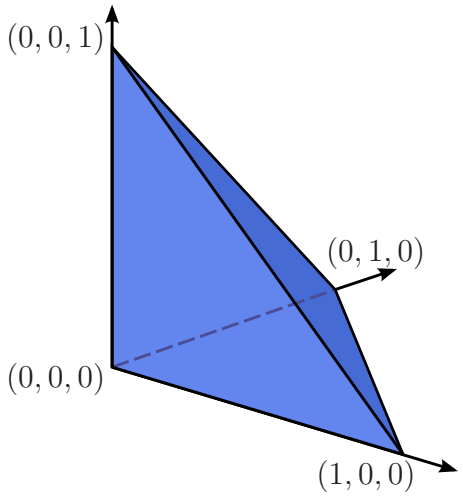

Figure 4.4: The reference tetrahedron.

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 4.6: Vertex coordinates of the reference hexahedron.

4.5 The reference hexahedron

The reference hexahedron is shown in Figure 4.5 and is defined by its eight vertices with coordinates as specified in Table 4.6.

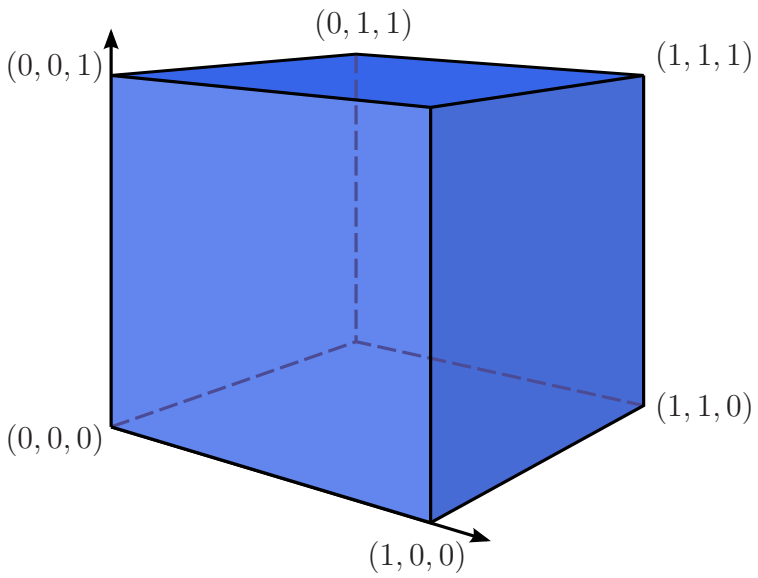

Figure 4.5: The reference hexahedron.

Chapter 5

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.

5.1 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 5.1.

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 $e_0 = (1, 0)$, $e_1 = (1, 1)$, $e_2 = (1, 2)$, $e_3 = (1, 3)$,

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

Table 5.1: Named mesh entities.

$e_4 = (1, 4)$ 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 $c_0 = (3, 0)$.

5.2 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 5.1 for a mesh consisting of two triangles.

For non-simplicial cells (quadrilaterals and hexahedra), the numbering is arbitrary, as long as each cell is isomorphic to the corresponding reference cell by matching each vertex with the corresponding vertex in the reference cell. This is illustrated in Figure 5.2 for a mesh consisting of two quadrilaterals.

5.3 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*.

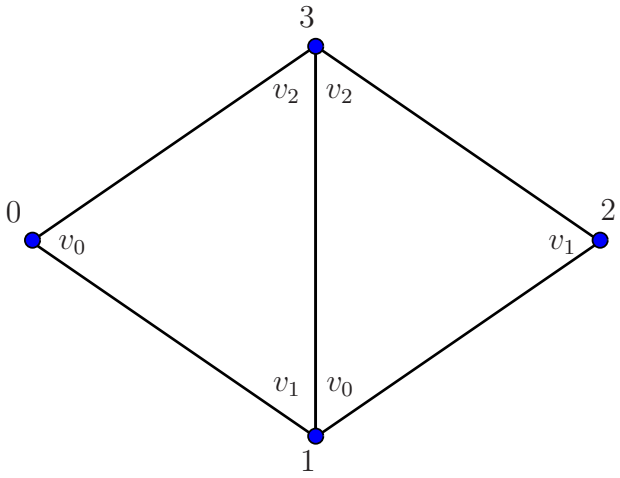

Figure 5.1: The vertices of a simplicial mesh are numbered locally based on the corresponding global vertex numbers.

As an illustration, consider the numbering of edges (the mesh entities of topological dimension one) on the reference triangle in Figure 5.3. 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 5.4.

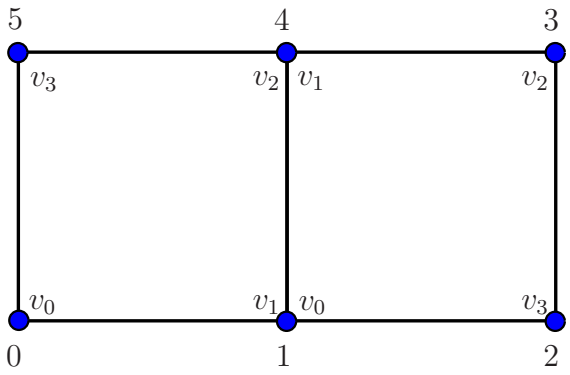

Figure 5.2: The local numbering of vertices of a non-simplicial mesh is arbitrary, as long as each cell is isomorphic to the reference cell by matching each vertex to the corresponding vertex of the reference cell.

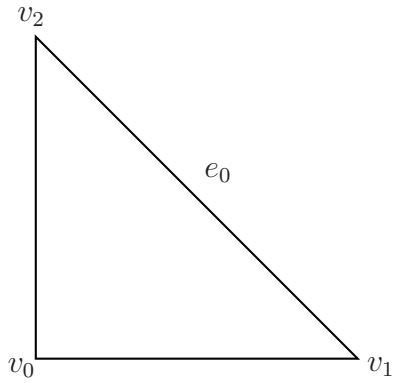

Figure 5.3: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertex v_0.

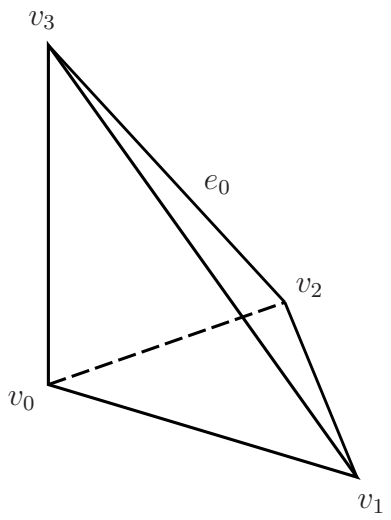

Figure 5.4: Mesh entities are ordered based on a lexicographical ordering of the corresponding ordered tuples of non-incident vertices. The first edge e_0 is non-incident to vertices v_0 and v_1.

5.3.1 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 5.5 for two incident triangles sharing a common edge.

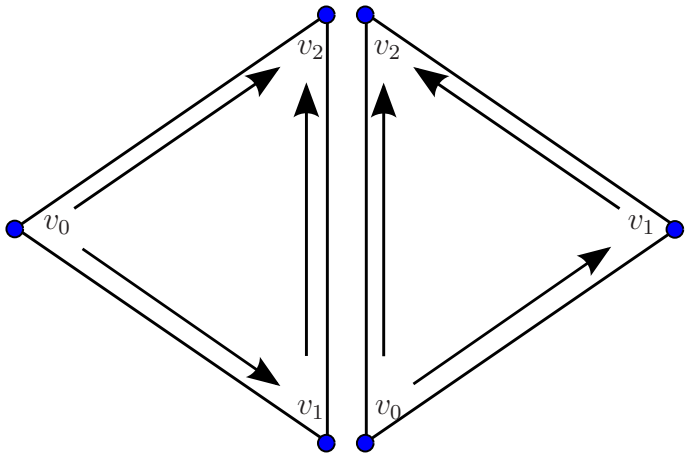

Figure 5.5: Two incident triangles will always agree on the orientation of the common edge.

5.3.2 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.

5.4 Numbering schemes for reference cells

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

5.4.1 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)	$\emptyset$

5.4.2 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)	$\emptyset$

5.4.3 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)	$\emptyset$

5.4.4 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)	$\emptyset$

5.4.5 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)
$c_0 = (3,0)$	$(v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$	$\emptyset$

Bibliography

[1] *Trilinos*. URL: http://software.sandia.gov/trilinos/.

[2] M. Alnæs and K.-A. Mardal, *SyFi*, 2007. URL: http://www.fenics.org/syfi/.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc*, 2006. URL: http://www.mcs.anl.gov/petsc/.

[4] W. Bangerth, R. Hartmann, and G. Kanschat, `deal.II` *Differential Equations Analysis Library*, 2006. URL: http://www.dealii.org/.

[5] A. M. Bruaset, H. P. Langtangen, et al., *Diffpack*, 2006. URL: http://www.diffpack.com/.

[6] P. G. Ciarlet, *The Finite Element Method for Elliptic Problems*, North-Holland, Amsterdam, New York, Oxford, 1978.

[7] J. Hoffman, J. Jansson, C. Johnson, M. G. Knepley, R. C. Kirby, A. Logg, L. R. Scott, and G. N. Wells, *FEniCS*, 2006. URL: http://www.fenics.org/.

[8] J. Hoffman, J. Jansson, A. Logg, and G. N. Wells, *DOLFIN*, 2006. URL: http://www.fenics.org/dolfin/.

[9] T. J. R. Hughes, *The Finite Element Method: Linear Static and Dynamic Finite Ele ment Analysis*, Prentice-Hall, 1987.

[10] R. C. Kirby, M. G. Knepley, A. Logg, and L. R. Scott, *Optimizing the evaluation of finite element matrices*, SIAM J. Sci. Comput., 27 (2005), pp. 741–758.

[11] R. C. Kirby and A. Logg, *A compiler for variational forms*, ACM Transactions on Mathematical Software, 32 (2006), pp. 417–444.

[12] ——, *Efficient compilation of a class of variational forms*, ACM Transactions on Mathematical Software, 33 (2007).

[13] R. C. Kirby, A. Logg, L. R. Scott, and A. R. Terrel, *Topological optimization of the evaluation of finite element ma trices*, SIAM J. Sci. Comput., 28 (2006), pp. 224–240.

[14] A. Logg, *FFC*, 2007. URL: http://www.fenics.org/ffc/.

[15] K. Long, *Sundance*, 2006. URL: http://software.sandia.gov/sundance/.

[16] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, *The Finite Element Method — Its Basis and Fundamentals, 6th e dition*, Elsevier, 2005, first published in 1967.

Appendix A

C++ Interface

Below follows a verbatim copy of the complete UFC interface which is specified in the header file `ufc.h`.

```
// This is UFC (Unified Form-assembly Code) v. 1.1.2.
// This code is released into the public domain.
//
// The FEniCS Project (http://www.fenics.org/) 2006-2009.

#ifndef __UFC_H
#define __UFC_H

#define UFC_VERSION_MAJOR 1
#define UFC_VERSION_MINOR 1
#define UFC_VERSION_MAINTENANCE 2

#include <stdexcept>

const char UFC_VERSION[] = "1.1";

namespace ufc
{

  /// Valid cell shapes
  enum shape {interval, triangle, quadrilateral, tetrahedron, hexahedron};

  /// This class defines the data structure for a finite element mesh.

  class mesh
  {
  public:

    /// Constructor
```

```
  mesh(): topological_dimension(0), geometric_dimension(0), num_entities(0) {}

  /// Destructor
  virtual ~mesh() {}

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

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

  /// Array of the global number of entities of each topological dimension
  unsigned int* num_entities;

};

/// This class defines the data structure for a cell in a mesh.

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;

};

/// This class defines the interface for a general tensor-valued function.

class function
{
public:

  /// Constructor
  function() {}

  /// Destructor
```

```
  virtual ~function() {}

  /// Evaluate function at given point in cell
  virtual void evaluate(double* values,
                        const double* coordinates,
                        const cell& c) const = 0;

};

/// This class defines the interface for a finite element.

class finite_element
{
public:

  /// Constructor
  finite_element() {}

  /// Destructor
  virtual ~finite_element() {}

  /// Return a string identifying the finite element
  virtual const char* signature() const = 0;

  /// Return the cell shape
  virtual shape cell_shape() const = 0;

  /// Return the dimension of the finite element function space
  virtual unsigned int space_dimension() const = 0;

  /// Return the rank of the value space
  virtual unsigned int value_rank() const = 0;

  /// Return the dimension of the value space for axis i
  virtual unsigned int value_dimension(unsigned int i) const = 0;

  /// Evaluate basis function i at given point in cell
  virtual void evaluate_basis(unsigned int i,
                              double* values,
                              const double* coordinates,
                              const cell& c) const = 0;

  /// Evaluate all basis functions at given point in cell
  virtual void evaluate_basis_all(double* values,
                                  const double* coordinates,
                                  const cell& c) const
  { throw std::runtime_error("Not implemented (introduced in UFC v1.1)."); }

  /// Evaluate order n derivatives of basis function i at given point in cell
  virtual void evaluate_basis_derivatives(unsigned int i,
                                          unsigned int n,
                                          double* values,
                                          const double* coordinates,
                                          const cell& c) const = 0;

  /// Evaluate order n derivatives of all basis functions at given point in cell
  virtual void evaluate_basis_derivatives_all(unsigned int n,
```

```
                                          double* values,
                                          const double* coordinates,
                                          const cell& c) const
  { throw std::runtime_error("Not implemented (introduced in UFC v1.1)."); }

  /// Evaluate linear functional for dof i on the function f
  virtual double evaluate_dof(unsigned int i,
                              const function& f,
                              const cell& c) const = 0;

  /// Evaluate linear functionals for all dofs on the function f
  virtual void evaluate_dofs(double* values,
                             const function& f,
                             const cell& c) const
  { throw std::runtime_error("Not implemented (introduced in UFC v1.1)."); }

  /// Interpolate vertex values from dof values
  virtual void interpolate_vertex_values(double* vertex_values,
                                         const double* dof_values,
                                         const cell& c) const = 0;

  /// Return the number of sub elements (for a mixed element)
  virtual unsigned int num_sub_elements() const = 0;

  /// Create a new finite element for sub element i (for a mixed element)
  virtual finite_element* create_sub_element(unsigned int i) const = 0;

};

/// This class defines the interface for a local-to-global mapping of
/// degrees of freedom (dofs).

class dof_map
{
public:

  /// Constructor
  dof_map() {}

  /// Destructor
  virtual ~dof_map() {}

  /// Return a string identifying the dof map
  virtual const char* signature() const = 0;

  /// Return true iff mesh entities of topological dimension d are needed
  virtual bool needs_mesh_entities(unsigned int d) const = 0;

  /// Initialize dof map for mesh (return true iff init_cell() is needed)
  virtual bool init_mesh(const mesh& mesh) = 0;

  /// Initialize dof map for given cell
  virtual void init_cell(const mesh& m,
                         const cell& c) = 0;

  /// Finish initialization of dof map for cells
  virtual void init_cell_finalize() = 0;
```

```
  /// Return the dimension of the global finite element function space
  virtual unsigned int global_dimension() const = 0;

  /// Return the dimension of the local finite element function space for a cell
  virtual unsigned int local_dimension(const cell& c) const = 0;

  /// Return the maximum dimension of the local finite element function space
  virtual unsigned int max_local_dimension() const = 0;

  // Return the geometric dimension of the coordinates this dof map provides
  virtual unsigned int geometric_dimension() const
  { throw std::runtime_error("Not implemented (introduced in UFC v1.1)."); }

  /// Return the number of dofs on each cell facet
  virtual unsigned int num_facet_dofs() const = 0;

  /// Return the number of dofs associated with each cell entity of dimension d
  virtual unsigned int num_entity_dofs(unsigned int d) const
  { throw std::runtime_error("Not implemented (introduced in UFC v1.1)."); }

  /// Tabulate the local-to-global mapping of dofs on a cell
  virtual void tabulate_dofs(unsigned int* dofs,
                             const mesh& m,
                             const cell& c) const = 0;

  /// Tabulate the local-to-local mapping from facet dofs to cell dofs
  virtual void tabulate_facet_dofs(unsigned int* dofs,
                                   unsigned int facet) const = 0;

  /// Tabulate the local-to-local mapping of dofs on entity (d, i)
  virtual void tabulate_entity_dofs(unsigned int* dofs,
                                    unsigned int d, unsigned int i) const
  { throw std::runtime_error("Not implemented (introduced in UFC v1.1)."); }

  /// Tabulate the coordinates of all dofs on a cell
  virtual void tabulate_coordinates(double** coordinates,
                                    const cell& c) const = 0;

  /// Return the number of sub dof maps (for a mixed element)
  virtual unsigned int num_sub_dof_maps() const = 0;

  /// Create a new dof_map for sub dof map i (for a mixed element)
  virtual dof_map* create_sub_dof_map(unsigned int i) const = 0;

};

/// This class defines the interface for the tabulation of the cell
/// tensor corresponding to the local contribution to a form from
/// the integral over a cell.

class cell_integral
{
public:

  /// Constructor
  cell_integral() {}
```

```
  /// Destructor
  virtual ~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;

};

/// This class defines the interface for the tabulation of the
/// exterior facet tensor corresponding to the local contribution to
/// a form from the integral over an exterior facet.

class exterior_facet_integral
{
public:

  /// Constructor
  exterior_facet_integral() {}

  /// Destructor
  virtual ~exterior_facet_integral() {}

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

};

/// This class defines the interface for the tabulation of the
/// interior facet tensor corresponding to the local contribution to
/// a form from the integral over an interior facet.

class interior_facet_integral
{
public:

  /// Constructor
  interior_facet_integral() {}

  /// Destructor
  virtual ~interior_facet_integral() {}

  /// Tabulate the tensor for the contribution from a local interior facet
  virtual void tabulate_tensor(double* A,
                               const double * const * w,
                               const cell& c0,
                               const cell& c1,
                               unsigned int facet0,
                               unsigned int facet1) const = 0;

};
```

```
/// This class defines the interface for the assembly of the global
/// tensor corresponding to a form with r + n arguments, that is, a
/// mapping
///
///     a : V1 x V2 x ... Vr x W1 x W2 x ... x Wn -> R
///
/// with arguments v1, v2, ..., vr, w1, w2, ..., wn. The rank r
/// global tensor A is defined by
///
///     A = a(V1, V2, ..., Vr, w1, w2, ..., wn),
///
/// where each argument Vj represents the application to the
/// sequence of basis functions of Vj and w1, w2, ..., wn are given
/// fixed functions (coefficients).

class form
{
public:

  /// Constructor
  form() {}

  /// Destructor
  virtual ~form() {}

  /// Return a string identifying the form
  virtual const char* signature() const = 0;

  /// Return the rank of the global tensor (r)
  virtual unsigned int rank() const = 0;

  /// Return the number of coefficients (n)
  virtual unsigned int num_coefficients() const = 0;

  /// Return the number of cell integrals
  virtual unsigned int num_cell_integrals() const = 0;

  /// Return the number of exterior facet integrals
  virtual unsigned int num_exterior_facet_integrals() const = 0;

  /// Return the number of interior facet integrals
  virtual unsigned int num_interior_facet_integrals() const = 0;

  /// Create a new finite element for argument function i
  virtual finite_element* create_finite_element(unsigned int i) const = 0;

  /// Create a new dof map for argument function i
  virtual dof_map* create_dof_map(unsigned int i) const = 0;

  /// 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;

  };

}

#endif
```

Appendix B

A basic UFC-based 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.[1]

```
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()];
```

[1] For an example of a complete implementation of a UFC-based assembler, we refer to the source code of DOLFIN [8], in particular class `Assembler` as implemented in `Assembler.cpp`.

```
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* AK = 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(AK, w, cell);

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

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

Appendix C

Complete UFC code for Poisson's equation

In this section, a simple example is given of UFC code generated by the form compilers FFC [14, 10, 13, 11, 12] and SyFi [2] for Poisson's equation. The code presented below is generated for the bilinear form of Poisson's equation for standard continuous piecewise linear Lagrange finite elements on a two-dimensional domain Ω,

$$a(v, u) = \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}x. \tag{C.1}$$

Although FFC and SyFi are two different form compilers, with very different approaches to code generation, both generate code that conforms to the UFC specification and may thus be used interchangeably within any UFC-based system.

In the generated code, we have omitted the two functions `evaluate_basis` and `evaluate_basis_derivatives`[1] to save space.

[1] For FFC, this may be done by using the compiler flags `-fno-evaluate_basis` and `-fno-evaluate_basis_derivatives`.

C.1 Code generated by FFC

```
// This code conforms with the UFC specification version 1.0
// and was automatically generated by FFC version 0.6.2.

#ifndef __POISSON_H
#define __POISSON_H

#include <cmath>
#include <stdexcept>
#include <ufc.h>

/// This class defines the interface for a finite element.

class PoissonBilinearForm_finite_element_0: public ufc::finite_element
{
public:

  /// Constructor
  PoissonBilinearForm_finite_element_0() : ufc::finite_element()
  {
    // Do nothing
  }

  /// Destructor
  virtual ~PoissonBilinearForm_finite_element_0()
  {
    // Do nothing
  }

  /// Return a string identifying the finite element
  virtual const char* signature() const
  {
    return "FiniteElement('Lagrange', 'triangle', 1)";
  }

  /// Return the cell shape
  virtual ufc::shape cell_shape() const
  {
    return ufc::triangle;
  }

  /// Return the dimension of the finite element function space
  virtual unsigned int space_dimension() const
  {
    return 3;
  }

  /// Return the rank of the value space
  virtual unsigned int value_rank() const
  {
    return 0;
  }

  /// Return the dimension of the value space for axis i
```

```
virtual unsigned int value_dimension(unsigned int i) const
{
  return 1;
}

/// Evaluate basis function i at given point in cell
virtual void evaluate_basis(unsigned int i,
                            double* values,
                            const double* coordinates,
                            const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * element_coordinates = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = element_coordinates[1][0] - element_coordinates[0][0];
  const double J_01 = element_coordinates[2][0] - element_coordinates[0][0];
  const double J_10 = element_coordinates[1][1] - element_coordinates[0][1];
  const double J_11 = element_coordinates[2][1] - element_coordinates[0][1];

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

  // Compute inverse of Jacobian

  // Get coordinates and map to the reference (UFC) element
  double x = (element_coordinates[0][1]*element_coordinates[2][0] -\
              element_coordinates[0][0]*element_coordinates[2][1] +\
              J_11*coordinates[0] - J_01*coordinates[1]) / detJ;
  double y = (element_coordinates[1][1]*element_coordinates[0][0] -\
              element_coordinates[1][0]*element_coordinates[0][1] -\
              J_10*coordinates[0] + J_00*coordinates[1]) / detJ;

  // Map coordinates to the reference square
  if (std::abs(y - 1.0) < 1e-14)
    x = -1.0;
  else
    x = 2.0 *x/(1.0 - y) - 1.0;
  y = 2.0*y - 1.0;

  // Reset values
  *values = 0;

  // Map degree of freedom to element degree of freedom
  const unsigned int dof = i;

  // Generate scalings
  const double scalings_y_0 = 1;
  const double scalings_y_1 = scalings_y_0*(0.5 - 0.5*y);

  // Compute psitilde_a
  const double psitilde_a_0 = 1;
  const double psitilde_a_1 = x;

  // Compute psitilde_bs
  const double psitilde_bs_0_0 = 1;
  const double psitilde_bs_0_1 = 1.5*y + 0.5;
```

```
  const double psitilde_bs_1_0 = 1;

  // Compute basisvalues
  const double basisvalue0 = 0.707106781186548*psitilde_a_0*scalings_y_0*psitilde_bs_0_0;
  const double basisvalue1 = 1.73205080756888*psitilde_a_1*scalings_y_1*psitilde_bs_1_0;
  const double basisvalue2 = psitilde_a_0*scalings_y_0*psitilde_bs_0_1;

  // Table(s) of coefficients
  const static double coefficients0[3][3] = \
  {{0.471404520791032, -0.288675134594813, -0.166666666666667},
  {0.471404520791032, 0.288675134594813, -0.166666666666667},
  {0.471404520791032, 0, 0.333333333333333}};

  // Extract relevant coefficients
  const double coeff0_0 = coefficients0[dof][0];
  const double coeff0_1 = coefficients0[dof][1];
  const double coeff0_2 = coefficients0[dof][2];

  // Compute value(s)
  *values = coeff0_0*basisvalue0 + coeff0_1*basisvalue1 + coeff0_2*basisvalue2;
}

/// Evaluate all basis functions at given point in cell
virtual void evaluate_basis_all(double* values,
                                const double* coordinates,
                                const ufc::cell& c) const
{
  throw std::runtime_error("The vectorised version of evaluate_basis() is not yet implemented.");
}

/// Evaluate order n derivatives of basis function i at given point in cell
virtual void evaluate_basis_derivatives(unsigned int i,
                                        unsigned int n,
                                        double* values,
                                        const double* coordinates,
                                        const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * element_coordinates = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = element_coordinates[1][0] - element_coordinates[0][0];
  const double J_01 = element_coordinates[2][0] - element_coordinates[0][0];
  const double J_10 = element_coordinates[1][1] - element_coordinates[0][1];
  const double J_11 = element_coordinates[2][1] - element_coordinates[0][1];

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

  // Compute inverse of Jacobian

  // Get coordinates and map to the reference (UFC) element
  double x = (element_coordinates[0][1]*element_coordinates[2][0] -\
              element_coordinates[0][0]*element_coordinates[2][1] +\
              J_11*coordinates[0] - J_01*coordinates[1]) / detJ;
  double y = (element_coordinates[1][1]*element_coordinates[0][0] -\
              element_coordinates[1][0]*element_coordinates[0][1] -\
```

```
            J_10*coordinates[0] + J_00*coordinates[1]) / detJ;

// Map coordinates to the reference square
if (std::abs(y - 1.0) < 1e-14)
  x = -1.0;
else
  x = 2.0 *x/(1.0 - y) - 1.0;
y = 2.0*y - 1.0;

// Compute number of derivatives
unsigned int num_derivatives = 1;

for (unsigned int j = 0; j < n; j++)
  num_derivatives *= 2;


// Declare pointer to two dimensional array that holds combinations of derivatives and initialise
unsigned int **combinations = new unsigned int *[num_derivatives];

for (unsigned int j = 0; j < num_derivatives; j++)
{
  combinations[j] = new unsigned int [n];
  for (unsigned int k = 0; k < n; k++)
    combinations[j][k] = 0;
}

// Generate combinations of derivatives
for (unsigned int row = 1; row < num_derivatives; row++)
{
  for (unsigned int num = 0; num < row; num++)
  {
    for (unsigned int col = n-1; col+1 > 0; col--)
    {
      if (combinations[row][col] + 1 > 1)
        combinations[row][col] = 0;
      else
      {
        combinations[row][col] += 1;
        break;
      }
    }
  }
}

// Compute inverse of Jacobian
const double Jinv[2][2] =  {{J_11 / detJ, -J_01 / detJ}, {-J_10 / detJ, J_00 / detJ}};

// Declare transformation matrix
// Declare pointer to two dimensional array and initialise
double **transform = new double *[num_derivatives];

for (unsigned int j = 0; j < num_derivatives; j++)
{
  transform[j] = new double [num_derivatives];
  for (unsigned int k = 0; k < num_derivatives; k++)
    transform[j][k] = 1;
}
```

```
// Construct transformation matrix
for (unsigned int row = 0; row < num_derivatives; row++)
{
  for (unsigned int col = 0; col < num_derivatives; col++)
  {
    for (unsigned int k = 0; k < n; k++)
      transform[row][col] *= Jinv[combinations[col][k]][combinations[row][k]];
  }
}

// Reset values
for (unsigned int j = 0; j < 1*num_derivatives; j++)
  values[j] = 0;

// Map degree of freedom to element degree of freedom
const unsigned int dof = i;

// Generate scalings
const double scalings_y_0 = 1;
const double scalings_y_1 = scalings_y_0*(0.5 - 0.5*y);

// Compute psitilde_a
const double psitilde_a_0 = 1;
const double psitilde_a_1 = x;

// Compute psitilde_bs
const double psitilde_bs_0_0 = 1;
const double psitilde_bs_0_1 = 1.5*y + 0.5;
const double psitilde_bs_1_0 = 1;

// Compute basisvalues
const double basisvalue0 = 0.707106781186548*psitilde_a_0*scalings_y_0*psitilde_bs_0_0;
const double basisvalue1 = 1.73205080756888*psitilde_a_1*scalings_y_1*psitilde_bs_1_0;
const double basisvalue2 = psitilde_a_0*scalings_y_0*psitilde_bs_0_1;

// Table(s) of coefficients
const static double coefficients0[3][3] = \
{{0.471404520791032, -0.288675134594813, -0.166666666666667},
{0.471404520791032, 0.288675134594813, -0.166666666666667},
{0.471404520791032, 0, 0.333333333333333}};

// Interesting (new) part
// Tables of derivatives of the polynomial base (transpose)
const static double dmats0[3][3] = \
{{0, 0, 0},
{4.89897948556636, 0, 0},
{0, 0, 0}};

const static double dmats1[3][3] = \
{{0, 0, 0},
{2.44948974278318, 0, 0},
{4.24264068711928, 0, 0}};

// Compute reference derivatives
// Declare pointer to array of derivatives on FIAT element
double *derivatives = new double [num_derivatives];
```

```
// Declare coefficients
double coeff0_0 = 0;
double coeff0_1 = 0;
double coeff0_2 = 0;

// Declare new coefficients
double new_coeff0_0 = 0;
double new_coeff0_1 = 0;
double new_coeff0_2 = 0;

// Loop possible derivatives
for (unsigned int deriv_num = 0; deriv_num < num_derivatives; deriv_num++)
{
  // Get values from coefficients array
  new_coeff0_0 = coefficients0[dof][0];
  new_coeff0_1 = coefficients0[dof][1];
  new_coeff0_2 = coefficients0[dof][2];

  // Loop derivative order
  for (unsigned int j = 0; j < n; j++)
  {
    // Update old coefficients
    coeff0_0 = new_coeff0_0;
    coeff0_1 = new_coeff0_1;
    coeff0_2 = new_coeff0_2;

    if(combinations[deriv_num][j] == 0)
    {
      new_coeff0_0 = coeff0_0*dmats0[0][0] + coeff0_1*dmats0[1][0] + coeff0_2*dmats0[2][0];
      new_coeff0_1 = coeff0_0*dmats0[0][1] + coeff0_1*dmats0[1][1] + coeff0_2*dmats0[2][1];
      new_coeff0_2 = coeff0_0*dmats0[0][2] + coeff0_1*dmats0[1][2] + coeff0_2*dmats0[2][2];
    }
    if(combinations[deriv_num][j] == 1)
    {
      new_coeff0_0 = coeff0_0*dmats1[0][0] + coeff0_1*dmats1[1][0] + coeff0_2*dmats1[2][0];
      new_coeff0_1 = coeff0_0*dmats1[0][1] + coeff0_1*dmats1[1][1] + coeff0_2*dmats1[2][1];
      new_coeff0_2 = coeff0_0*dmats1[0][2] + coeff0_1*dmats1[1][2] + coeff0_2*dmats1[2][2];
    }

  }
  // Compute derivatives on reference element as dot product of coefficients and basisvalues
  derivatives[deriv_num] = new_coeff0_0*basisvalue0 + new_coeff0_1*basisvalue1 + new_coeff0_2*basisvalue2;
}

// Transform derivatives back to physical element
for (unsigned int row = 0; row < num_derivatives; row++)
{
  for (unsigned int col = 0; col < num_derivatives; col++)
  {
    values[row] += transform[row][col]*derivatives[col];
  }
}
// Delete pointer to array of derivatives on FIAT element
delete [] derivatives;

// Delete pointer to array of combinations of derivatives and transform
```

```
  for (unsigned int row = 0; row < num_derivatives; row++)
  {
    delete [] combinations[row];
    delete [] transform[row];
  }

  delete [] combinations;
  delete [] transform;
}

/// Evaluate order n derivatives of all basis functions at given point in cell
virtual void evaluate_basis_derivatives_all(unsigned int n,
                                            double* values,
                                            const double* coordinates,
                                            const ufc::cell& c) const
{
  throw std::runtime_error("The vectorised version of evaluate_basis_derivatives() is not yet implemented.");
}

/// Evaluate linear functional for dof i on the function f
virtual double evaluate_dof(unsigned int i,
                            const ufc::function& f,
                            const ufc::cell& c) const
{
  // The reference points, direction and weights:
  const static double X[3][1][2] = {{{0, 0}}, {{1, 0}}, {{0, 1}}};
  const static double W[3][1] = {{1}, {1}, {1}};
  const static double D[3][1][1] = {{{1}}, {{1}}, {{1}}};

  const double * const * x = c.coordinates;
  double result = 0.0;
  // Iterate over the points:
  // Evaluate basis functions for affine mapping
  const double w0 = 1.0 - X[i][0][0] - X[i][0][1];
  const double w1 = X[i][0][0];
  const double w2 = X[i][0][1];

  // Compute affine mapping y = F(X)
  double y[2];
  y[0] = w0*x[0][0] + w1*x[1][0] + w2*x[2][0];
  y[1] = w0*x[0][1] + w1*x[1][1] + w2*x[2][1];

  // Evaluate function at physical points
  double values[1];
  f.evaluate(values, y, c);

  // Map function values using appropriate mapping
  // Affine map: Do nothing

  // Note that we do not map the weights (yet).

  // Take directional components
  for(int k = 0; k < 1; k++)
    result += values[k]*D[i][0][k];
  // Multiply by weights
  result *= W[i][0];
```

```
    return result;
  }

  /// Evaluate linear functionals for all dofs on the function f
  virtual void evaluate_dofs(double* values,
                             const ufc::function& f,
                             const ufc::cell& c) const
  {
    throw std::runtime_error("Not implemented (introduced in UFC v1.1).");
  }

  /// Interpolate vertex values from dof values
  virtual void interpolate_vertex_values(double* vertex_values,
                                         const double* dof_values,
                                         const ufc::cell& c) const
  {
    // Evaluate at vertices and use affine mapping
    vertex_values[0] = dof_values[0];
    vertex_values[1] = dof_values[1];
    vertex_values[2] = dof_values[2];
  }

  /// Return the number of sub elements (for a mixed element)
  virtual unsigned int num_sub_elements() const
  {
    return 1;
  }

  /// Create a new finite element for sub element i (for a mixed element)
  virtual ufc::finite_element* create_sub_element(unsigned int i) const
  {
    return new PoissonBilinearForm_finite_element_0();
  }

};

/// This class defines the interface for a finite element.

class PoissonBilinearForm_finite_element_1: public ufc::finite_element
{
public:

  /// Constructor
  PoissonBilinearForm_finite_element_1() : ufc::finite_element()
  {
    // Do nothing
  }

  /// Destructor
  virtual ~PoissonBilinearForm_finite_element_1()
  {
    // Do nothing
  }

  /// Return a string identifying the finite element
  virtual const char* signature() const
  {
```

```
  return "FiniteElement('Lagrange', 'triangle', 1)";
}

/// Return the cell shape
virtual ufc::shape cell_shape() const
{
  return ufc::triangle;
}

/// Return the dimension of the finite element function space
virtual unsigned int space_dimension() const
{
  return 3;
}

/// Return the rank of the value space
virtual unsigned int value_rank() const
{
  return 0;
}

/// Return the dimension of the value space for axis i
virtual unsigned int value_dimension(unsigned int i) const
{
  return 1;
}

/// Evaluate basis function i at given point in cell
virtual void evaluate_basis(unsigned int i,
                            double* values,
                            const double* coordinates,
                            const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * element_coordinates = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = element_coordinates[1][0] - element_coordinates[0][0];
  const double J_01 = element_coordinates[2][0] - element_coordinates[0][0];
  const double J_10 = element_coordinates[1][1] - element_coordinates[0][1];
  const double J_11 = element_coordinates[2][1] - element_coordinates[0][1];

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

  // Compute inverse of Jacobian

  // Get coordinates and map to the reference (UFC) element
  double x = (element_coordinates[0][1]*element_coordinates[2][0] -\
              element_coordinates[0][0]*element_coordinates[2][1] +\
              J_11*coordinates[0] - J_01*coordinates[1]) / detJ;
  double y = (element_coordinates[1][1]*element_coordinates[0][0] -\
              element_coordinates[1][0]*element_coordinates[0][1] -\
              J_10*coordinates[0] + J_00*coordinates[1]) / detJ;

  // Map coordinates to the reference square
  if (std::abs(y - 1.0) < 1e-14)
```

```
    x = -1.0;
  else
    x = 2.0 *x/(1.0 - y) - 1.0;
  y = 2.0*y - 1.0;

  // Reset values
  *values = 0;

  // Map degree of freedom to element degree of freedom
  const unsigned int dof = i;

  // Generate scalings
  const double scalings_y_0 = 1;
  const double scalings_y_1 = scalings_y_0*(0.5 - 0.5*y);

  // Compute psitilde_a
  const double psitilde_a_0 = 1;
  const double psitilde_a_1 = x;

  // Compute psitilde_bs
  const double psitilde_bs_0_0 = 1;
  const double psitilde_bs_0_1 = 1.5*y + 0.5;
  const double psitilde_bs_1_0 = 1;

  // Compute basisvalues
  const double basisvalue0 = 0.707106781186548*psitilde_a_0*scalings_y_0*psitilde_bs_0_0;
  const double basisvalue1 = 1.73205080756888*psitilde_a_1*scalings_y_1*psitilde_bs_1_0;
  const double basisvalue2 = psitilde_a_0*scalings_y_0*psitilde_bs_0_1;

  // Table(s) of coefficients
  const static double coefficients0[3][3] = \
  {{0.471404520791032, -0.288675134594813, -0.166666666666667},
  {0.471404520791032, 0.288675134594813, -0.166666666666667},
  {0.471404520791032, 0, 0.333333333333333}};

  // Extract relevant coefficients
  const double coeff0_0 = coefficients0[dof][0];
  const double coeff0_1 = coefficients0[dof][1];
  const double coeff0_2 = coefficients0[dof][2];

  // Compute value(s)
  *values = coeff0_0*basisvalue0 + coeff0_1*basisvalue1 + coeff0_2*basisvalue2;
}

/// Evaluate all basis functions at given point in cell
virtual void evaluate_basis_all(double* values,
                                const double* coordinates,
                                const ufc::cell& c) const
{
  throw std::runtime_error("The vectorised version of evaluate_basis() is not yet implemented.");
}

/// Evaluate order n derivatives of basis function i at given point in cell
virtual void evaluate_basis_derivatives(unsigned int i,
                                        unsigned int n,
                                        double* values,
                                        const double* coordinates,
```

```
                                        const ufc::cell& c) const
{
  // Extract vertex coordinates
  const double * const * element_coordinates = c.coordinates;

  // Compute Jacobian of affine map from reference cell
  const double J_00 = element_coordinates[1][0] - element_coordinates[0][0];
  const double J_01 = element_coordinates[2][0] - element_coordinates[0][0];
  const double J_10 = element_coordinates[1][1] - element_coordinates[0][1];
  const double J_11 = element_coordinates[2][1] - element_coordinates[0][1];

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

  // Compute inverse of Jacobian

  // Get coordinates and map to the reference (UFC) element
  double x = (element_coordinates[0][1]*element_coordinates[2][0] -\
              element_coordinates[0][0]*element_coordinates[2][1] +\
              J_11*coordinates[0] - J_01*coordinates[1]) / detJ;
  double y = (element_coordinates[1][1]*element_coordinates[0][0] -\
              element_coordinates[1][0]*element_coordinates[0][1] -\
              J_10*coordinates[0] + J_00*coordinates[1]) / detJ;

  // Map coordinates to the reference square
  if (std::abs(y - 1.0) < 1e-14)
    x = -1.0;
  else
    x = 2.0 *x/(1.0 - y) - 1.0;
  y = 2.0*y - 1.0;

  // Compute number of derivatives
  unsigned int num_derivatives = 1;

  for (unsigned int j = 0; j < n; j++)
    num_derivatives *= 2;


  // Declare pointer to two dimensional array that holds combinations of derivatives and initialise
  unsigned int **combinations = new unsigned int *[num_derivatives];

  for (unsigned int j = 0; j < num_derivatives; j++)
  {
    combinations[j] = new unsigned int [n];
    for (unsigned int k = 0; k < n; k++)
      combinations[j][k] = 0;
  }

  // Generate combinations of derivatives
  for (unsigned int row = 1; row < num_derivatives; row++)
  {
    for (unsigned int num = 0; num < row; num++)
    {
      for (unsigned int col = n-1; col+1 > 0; col--)
      {
        if (combinations[row][col] + 1 > 1)
          combinations[row][col] = 0;
```

```
      else
      {
        combinations[row][col] += 1;
        break;
      }
    }
  }
}

// Compute inverse of Jacobian
const double Jinv[2][2] =  {{J_11 / detJ, -J_01 / detJ}, {-J_10 / detJ, J_00 / detJ}};

// Declare transformation matrix
// Declare pointer to two dimensional array and initialise
double **transform = new double *[num_derivatives];

for (unsigned int j = 0; j < num_derivatives; j++)
{
  transform[j] = new double [num_derivatives];
  for (unsigned int k = 0; k < num_derivatives; k++)
    transform[j][k] = 1;
}

// Construct transformation matrix
for (unsigned int row = 0; row < num_derivatives; row++)
{
  for (unsigned int col = 0; col < num_derivatives; col++)
  {
    for (unsigned int k = 0; k < n; k++)
      transform[row][col] *= Jinv[combinations[col][k]][combinations[row][k]];
  }
}

// Reset values
for (unsigned int j = 0; j < 1*num_derivatives; j++)
  values[j] = 0;

// Map degree of freedom to element degree of freedom
const unsigned int dof = i;

// Generate scalings
const double scalings_y_0 = 1;
const double scalings_y_1 = scalings_y_0*(0.5 - 0.5*y);

// Compute psitilde_a
const double psitilde_a_0 = 1;
const double psitilde_a_1 = x;

// Compute psitilde_bs
const double psitilde_bs_0_0 = 1;
const double psitilde_bs_0_1 = 1.5*y + 0.5;
const double psitilde_bs_1_0 = 1;

// Compute basisvalues
const double basisvalue0 = 0.707106781186548*psitilde_a_0*scalings_y_0*psitilde_bs_0_0;
const double basisvalue1 = 1.73205080756888*psitilde_a_1*scalings_y_1*psitilde_bs_1_0;
const double basisvalue2 = psitilde_a_0*scalings_y_0*psitilde_bs_0_1;
```

```
// Table(s) of coefficients
const static double coefficients0[3][3] = \
{{0.471404520791032, -0.288675134594813, -0.166666666666667},
{0.471404520791032, 0.288675134594813, -0.166666666666667},
{0.471404520791032, 0, 0.333333333333333}};

// Interesting (new) part
// Tables of derivatives of the polynomial base (transpose)
const static double dmats0[3][3] = \
{{0, 0, 0},
{4.89897948556636, 0, 0},
{0, 0, 0}};

const static double dmats1[3][3] = \
{{0, 0, 0},
{2.44948974278318, 0, 0},
{4.24264068711928, 0, 0}};

// Compute reference derivatives
// Declare pointer to array of derivatives on FIAT element
double *derivatives = new double [num_derivatives];

// Declare coefficients
double coeff0_0 = 0;
double coeff0_1 = 0;
double coeff0_2 = 0;

// Declare new coefficients
double new_coeff0_0 = 0;
double new_coeff0_1 = 0;
double new_coeff0_2 = 0;

// Loop possible derivatives
for (unsigned int deriv_num = 0; deriv_num < num_derivatives; deriv_num++)
{
  // Get values from coefficients array
  new_coeff0_0 = coefficients0[dof][0];
  new_coeff0_1 = coefficients0[dof][1];
  new_coeff0_2 = coefficients0[dof][2];

  // Loop derivative order
  for (unsigned int j = 0; j < n; j++)
  {
    // Update old coefficients
    coeff0_0 = new_coeff0_0;
    coeff0_1 = new_coeff0_1;
    coeff0_2 = new_coeff0_2;

    if(combinations[deriv_num][j] == 0)
    {
      new_coeff0_0 = coeff0_0*dmats0[0][0] + coeff0_1*dmats0[1][0] + coeff0_2*dmats0[2][0];
      new_coeff0_1 = coeff0_0*dmats0[0][1] + coeff0_1*dmats0[1][1] + coeff0_2*dmats0[2][1];
      new_coeff0_2 = coeff0_0*dmats0[0][2] + coeff0_1*dmats0[1][2] + coeff0_2*dmats0[2][2];
    }
    if(combinations[deriv_num][j] == 1)
    {
```

```
        new_coeff0_0 = coeff0_0*dmats1[0][0] + coeff0_1*dmats1[1][0] + coeff0_2*dmats1[2][0];
        new_coeff0_1 = coeff0_0*dmats1[0][1] + coeff0_1*dmats1[1][1] + coeff0_2*dmats1[2][1];
        new_coeff0_2 = coeff0_0*dmats1[0][2] + coeff0_1*dmats1[1][2] + coeff0_2*dmats1[2][2];
      }

    }
    // Compute derivatives on reference element as dot product of coefficients and basisvalues
    derivatives[deriv_num] = new_coeff0_0*basisvalue0 + new_coeff0_1*basisvalue1 + new_coeff0_2*basisvalue2;
  }

  // Transform derivatives back to physical element
  for (unsigned int row = 0; row < num_derivatives; row++)
  {
    for (unsigned int col = 0; col < num_derivatives; col++)
    {
      values[row] += transform[row][col]*derivatives[col];
    }
  }
  // Delete pointer to array of derivatives on FIAT element
  delete [] derivatives;

  // Delete pointer to array of combinations of derivatives and transform
  for (unsigned int row = 0; row < num_derivatives; row++)
  {
    delete [] combinations[row];
    delete [] transform[row];
  }

  delete [] combinations;
  delete [] transform;
}

/// Evaluate order n derivatives of all basis functions at given point in cell
virtual void evaluate_basis_derivatives_all(unsigned int n,
                                            double* values,
                                            const double* coordinates,
                                            const ufc::cell& c) const
{
  throw std::runtime_error("The vectorised version of evaluate_basis_derivatives() is not yet implemented.");
}

/// Evaluate linear functional for dof i on the function f
virtual double evaluate_dof(unsigned int i,
                            const ufc::function& f,
                            const ufc::cell& c) const
{
  // The reference points, direction and weights:
  const static double X[3][1][2] = {{{0, 0}}, {{1, 0}}, {{0, 1}}};
  const static double W[3][1] = {{1}, {1}, {1}};
  const static double D[3][1][1] = {{{1}}, {{1}}, {{1}}};

  const double * const * x = c.coordinates;
  double result = 0.0;
  // Iterate over the points:
  // Evaluate basis functions for affine mapping
  const double w0 = 1.0 - X[i][0][0] - X[i][0][1];
  const double w1 = X[i][0][0];
```

```
    const double w2 = X[i][0][1];

    // Compute affine mapping y = F(X)
    double y[2];
    y[0] = w0*x[0][0] + w1*x[1][0] + w2*x[2][0];
    y[1] = w0*x[0][1] + w1*x[1][1] + w2*x[2][1];

    // Evaluate function at physical points
    double values[1];
    f.evaluate(values, y, c);

    // Map function values using appropriate mapping
    // Affine map: Do nothing

    // Note that we do not map the weights (yet).

    // Take directional components
    for(int k = 0; k < 1; k++)
      result += values[k]*D[i][0][k];
    // Multiply by weights
    result *= W[i][0];

    return result;
  }

  /// Evaluate linear functionals for all dofs on the function f
  virtual void evaluate_dofs(double* values,
                             const ufc::function& f,
                             const ufc::cell& c) const
  {
    throw std::runtime_error("Not implemented (introduced in UFC v1.1).");
  }

  /// Interpolate vertex values from dof values
  virtual void interpolate_vertex_values(double* vertex_values,
                                         const double* dof_values,
                                         const ufc::cell& c) const
  {
    // Evaluate at vertices and use affine mapping
    vertex_values[0] = dof_values[0];
    vertex_values[1] = dof_values[1];
    vertex_values[2] = dof_values[2];
  }

  /// Return the number of sub elements (for a mixed element)
  virtual unsigned int num_sub_elements() const
  {
    return 1;
  }

  /// Create a new finite element for sub element i (for a mixed element)
  virtual ufc::finite_element* create_sub_element(unsigned int i) const
  {
    return new PoissonBilinearForm_finite_element_1();
  }

};
```

```
/// This class defines the interface for a local-to-global mapping of
/// degrees of freedom (dofs).

class PoissonBilinearForm_dof_map_0: public ufc::dof_map
{
private:

  unsigned int __global_dimension;

public:

  /// Constructor
  PoissonBilinearForm_dof_map_0() : ufc::dof_map()
  {
    __global_dimension = 0;
  }

  /// Destructor
  virtual ~PoissonBilinearForm_dof_map_0()
  {
    // Do nothing
  }

  /// Return a string identifying the dof map
  virtual const char* signature() const
  {
    return "FFC dof map for FiniteElement('Lagrange', 'triangle', 1)";
  }

  /// Return true iff mesh entities of topological dimension d are needed
  virtual bool needs_mesh_entities(unsigned int d) const
  {
    switch (d)
    {
    case 0:
      return true;
      break;
    case 1:
      return false;
      break;
    case 2:
      return false;
      break;
    }
    return false;
  }

  /// Initialize dof map for mesh (return true iff init_cell() is needed)
  virtual bool init_mesh(const ufc::mesh& m)
  {
    __global_dimension = m.num_entities[0];
    return false;
  }

  /// Initialize dof map for given cell
  virtual void init_cell(const ufc::mesh& m,
```

```
                        const ufc::cell& c)
{
  // Do nothing
}

/// Finish initialization of dof map for cells
virtual void init_cell_finalize()
{
  // Do nothing
}

/// Return the dimension of the global finite element function space
virtual unsigned int global_dimension() const
{
  return __global_dimension;
}

/// Return the dimension of the local finite element function space
virtual unsigned int local_dimension() const
{
  return 3;
}

// Return the geometric dimension of the coordinates this dof map provides
virtual unsigned int geometric_dimension() const
{
  return 2;
}

/// Return the number of dofs on each cell facet
virtual unsigned int num_facet_dofs() const
{
  return 2;
}

/// Return the number of dofs associated with each cell entity of dimension d
virtual unsigned int num_entity_dofs(unsigned int d) const
{
  throw std::runtime_error("Not implemented (introduced in UFC v1.1).");
}

/// Tabulate the local-to-global mapping of dofs on a cell
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];
}

/// Tabulate the local-to-local mapping from facet dofs to cell dofs
virtual void tabulate_facet_dofs(unsigned int* dofs,
                                 unsigned int facet) const
{
  switch (facet)
  {
```

```
    case 0:
      dofs[0] = 1;
      dofs[1] = 2;
      break;
    case 1:
      dofs[0] = 0;
      dofs[1] = 2;
      break;
    case 2:
      dofs[0] = 0;
      dofs[1] = 1;
      break;
    }
  }

  /// Tabulate the local-to-local mapping of dofs on entity (d, i)
  virtual void tabulate_entity_dofs(unsigned int* dofs,
                                    unsigned int d, unsigned int i) const
  {
    throw std::runtime_error("Not implemented (introduced in UFC v1.1).");
  }

  /// Tabulate the coordinates of all dofs on a cell
  virtual void tabulate_coordinates(double** coordinates,
                                    const ufc::cell& c) const
  {
    const double * const * x = c.coordinates;
    coordinates[0][0] = x[0][0];
    coordinates[0][1] = x[0][1];
    coordinates[1][0] = x[1][0];
    coordinates[1][1] = x[1][1];
    coordinates[2][0] = x[2][0];
    coordinates[2][1] = x[2][1];
  }

  /// Return the number of sub dof maps (for a mixed element)
  virtual unsigned int num_sub_dof_maps() const
  {
    return 1;
  }

  /// Create a new dof_map for sub dof map i (for a mixed element)
  virtual ufc::dof_map* create_sub_dof_map(unsigned int i) const
  {
    return new PoissonBilinearForm_dof_map_0();
  }

};

/// This class defines the interface for a local-to-global mapping of
/// degrees of freedom (dofs).

class PoissonBilinearForm_dof_map_1: public ufc::dof_map
{
private:

  unsigned int __global_dimension;
```

```
public:

  /// Constructor
  PoissonBilinearForm_dof_map_1() : ufc::dof_map()
  {
    __global_dimension = 0;
  }

  /// Destructor
  virtual ~PoissonBilinearForm_dof_map_1()
  {
    // Do nothing
  }

  /// Return a string identifying the dof map
  virtual const char* signature() const
  {
    return "FFC dof map for FiniteElement('Lagrange', 'triangle', 1)";
  }

  /// Return true iff mesh entities of topological dimension d are needed
  virtual bool needs_mesh_entities(unsigned int d) const
  {
    switch (d)
    {
    case 0:
      return true;
      break;
    case 1:
      return false;
      break;
    case 2:
      return false;
      break;
    }
    return false;
  }

  /// Initialize dof map for mesh (return true iff init_cell() is needed)
  virtual bool init_mesh(const ufc::mesh& m)
  {
    __global_dimension = m.num_entities[0];
    return false;
  }

  /// Initialize dof map for given cell
  virtual void init_cell(const ufc::mesh& m,
                         const ufc::cell& c)
  {
    // Do nothing
  }

  /// Finish initialization of dof map for cells
  virtual void init_cell_finalize()
  {
    // Do nothing
```

```
}

/// Return the dimension of the global finite element function space
virtual unsigned int global_dimension() const
{
  return __global_dimension;
}

/// Return the dimension of the local finite element function space
virtual unsigned int local_dimension() const
{
  return 3;
}

// Return the geometric dimension of the coordinates this dof map provides
virtual unsigned int geometric_dimension() const
{
  return 2;
}

/// Return the number of dofs on each cell facet
virtual unsigned int num_facet_dofs() const
{
  return 2;
}

/// Return the number of dofs associated with each cell entity of dimension d
virtual unsigned int num_entity_dofs(unsigned int d) const
{
  throw std::runtime_error("Not implemented (introduced in UFC v1.1).");
}

/// Tabulate the local-to-global mapping of dofs on a cell
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];
}

/// Tabulate the local-to-local mapping from facet dofs to cell dofs
virtual void tabulate_facet_dofs(unsigned int* dofs,
                                 unsigned int facet) const
{
  switch (facet)
  {
  case 0:
    dofs[0] = 1;
    dofs[1] = 2;
    break;
  case 1:
    dofs[0] = 0;
    dofs[1] = 2;
    break;
  case 2:
```

```
      dofs[0] = 0;
      dofs[1] = 1;
      break;
    }
  }

  /// Tabulate the local-to-local mapping of dofs on entity (d, i)
  virtual void tabulate_entity_dofs(unsigned int* dofs,
                                    unsigned int d, unsigned int i) const
  {
    throw std::runtime_error("Not implemented (introduced in UFC v1.1).");
  }

  /// Tabulate the coordinates of all dofs on a cell
  virtual void tabulate_coordinates(double** coordinates,
                                    const ufc::cell& c) const
  {
    const double * const * x = c.coordinates;
    coordinates[0][0] = x[0][0];
    coordinates[0][1] = x[0][1];
    coordinates[1][0] = x[1][0];
    coordinates[1][1] = x[1][1];
    coordinates[2][0] = x[2][0];
    coordinates[2][1] = x[2][1];
  }

  /// Return the number of sub dof maps (for a mixed element)
  virtual unsigned int num_sub_dof_maps() const
  {
    return 1;
  }

  /// Create a new dof_map for sub dof map i (for a mixed element)
  virtual ufc::dof_map* create_sub_dof_map(unsigned int i) const
  {
    return new PoissonBilinearForm_dof_map_1();
  }

};

/// This class defines the interface for the tabulation of the cell
/// tensor corresponding to the local contribution to a form from
/// the integral over a cell.

class PoissonBilinearForm_cell_integral_0_quadrature: public ufc::cell_integral
{
public:

  /// Constructor
  PoissonBilinearForm_cell_integral_0_quadrature() : ufc::cell_integral()
  {
    // Do nothing
  }

  /// Destructor
  virtual ~PoissonBilinearForm_cell_integral_0_quadrature()
  {
```

```
  // Do nothing
}

/// 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
{
  // 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 Jinv_00 =  J_11 / detJ;
  const double Jinv_01 = -J_01 / detJ;
  const double Jinv_10 = -J_10 / detJ;
  const double Jinv_11 =  J_00 / detJ;

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


  // Array of quadrature weights
  const static double W1 = 0.5;


  const static double FE0_D10[1][3] = \
  {{-1, 1, 0}};

  const static double FE0_D01[1][3] = \
  {{-1, 0, 1}};

  // Compute element tensor using UFL quadrature representation
  // Optimisations: ('simplify expressions', False), ('ignore zero tables', False), ('non zero columns', False), ('remove
  // Total number of operations to compute element tensor: 162

  // Loop quadrature points for integral
  // Number of operations to compute element tensor for following IP loop = 162
  // Only 1 integration point, omitting IP loop.

  // Number of operations for primary indices = 162
  for (unsigned int j = 0; j < 3; j++)
  {
    for (unsigned int k = 0; k < 3; k++)
    {
      // Number of operations to compute entry = 18
      A[j*3 + k] += ((Jinv_00*FE0_D10[0][j] + Jinv_10*FE0_D01[0][j])*(Jinv_00*FE0_D10[0][k] + Jinv_10*FE0_D01[0][k]) + (J
    }// end loop over 'k'
  }// end loop over 'j'
```

```
  }

};

/// This class defines the interface for the tabulation of the cell
/// tensor corresponding to the local contribution to a form from
/// the integral over a cell.

class PoissonBilinearForm_cell_integral_0: public ufc::cell_integral
{
private:

  PoissonBilinearForm_cell_integral_0_quadrature integral_0_quadrature;

public:

  /// Constructor
  PoissonBilinearForm_cell_integral_0() : ufc::cell_integral()
  {
    // Do nothing
  }

  /// Destructor
  virtual ~PoissonBilinearForm_cell_integral_0()
  {
    // Do nothing
  }

  /// 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
  {
    // Reset values of the element tensor block
    A[0] = 0;
    A[1] = 0;
    A[2] = 0;
    A[3] = 0;
    A[4] = 0;
    A[5] = 0;
    A[6] = 0;
    A[7] = 0;
    A[8] = 0;

    // Add all contributions to element tensor
    integral_0_quadrature.tabulate_tensor(A, w, c);
  }

};

/// This class defines the interface for the assembly of the global
/// tensor corresponding to a form with r + n arguments, that is, a
/// mapping
///
///     a : V1 x V2 x ... Vr x W1 x W2 x ... x Wn -> R
///
/// with arguments v1, v2, ..., vr, w1, w2, ..., wn. The rank r
```

```
/// global tensor A is defined by
///
///     A = a(V1, V2, ..., Vr, w1, w2, ..., wn),
///
/// where each argument Vj represents the application to the
/// sequence of basis functions of Vj and w1, w2, ..., wn are given
/// fixed functions (coefficients).

class PoissonBilinearForm: public ufc::form
{
public:

  /// Constructor
  PoissonBilinearForm() : ufc::form()
  {
    // Do nothing
  }

  /// Destructor
  virtual ~PoissonBilinearForm()
  {
    // Do nothing
  }

  /// Return a string identifying the form
  virtual const char* signature() const
  {
    return "Form([Integral(IndexSum(Product(Indexed(ComponentTensor(SpatialDerivative(BasisFunction(FiniteElement('Lagrange
  }

  /// Return the rank of the global tensor (r)
  virtual unsigned int rank() const
  {
    return 2;
  }

  /// Return the number of coefficients (n)
  virtual unsigned int num_coefficients() const
  {
    return 0;
  }

  /// Return the number of cell integrals
  virtual unsigned int num_cell_integrals() const
  {
    return 1;
  }

  /// Return the number of exterior facet integrals
  virtual unsigned int num_exterior_facet_integrals() const
  {
    return 0;
  }

  /// Return the number of interior facet integrals
  virtual unsigned int num_interior_facet_integrals() const
  {
```

```
    return 0;
  }

  /// Create a new finite element for argument function i
  virtual ufc::finite_element* create_finite_element(unsigned int i) const
  {
    switch (i)
    {
    case 0:
      return new PoissonBilinearForm_finite_element_0();
      break;
    case 1:
      return new PoissonBilinearForm_finite_element_1();
      break;
    }
    return 0;
  }

  /// Create a new dof map for argument function i
  virtual ufc::dof_map* create_dof_map(unsigned int i) const
  {
    switch (i)
    {
    case 0:
      return new PoissonBilinearForm_dof_map_0();
      break;
    case 1:
      return new PoissonBilinearForm_dof_map_1();
      break;
    }
    return 0;
  }

  /// Create a new cell integral on sub domain i
  virtual ufc::cell_integral* create_cell_integral(unsigned int i) const
  {
    return new PoissonBilinearForm_cell_integral_0();
  }

  /// Create a new exterior facet integral on sub domain i
  virtual ufc::exterior_facet_integral* create_exterior_facet_integral(unsigned int i) const
  {
    return 0;
  }

  /// Create a new interior facet integral on sub domain i
  virtual ufc::interior_facet_integral* create_interior_facet_integral(unsigned int i) const
  {
    return 0;
  }

};

#endif
```

C.2 Code generated by SyFi

In the following we list the complete code for the finite element, the dofmap and the variational form for computing a stiffness matrix based on linear Lagrangian elements in 2D.

The code below is generated for the assembler in PyCC and it therefore includes some PyCC files, since the option `SFC.options.include_from = "pycc"` was used during the code generation. If PyCC is not present, the option `SFC.options.include_from = "syfi"` can be used, and this will result in some additional files that define the numbering scheme.

C.2.1 Header file for linear Lagrange element in 2D

```
//
// This code complies with UFC version 1.0, and is generated with SyFi version 0.4.0.
//
// http://www.fenics.org/syfi/
// http://www.fenics.org/ufc/
//


#ifndef __fe_Lagrange_1_2D_H
#define __fe_Lagrange_1_2D_H

#include <stdexcept>
#include <math.h>
#include <ufc.h>
#include <pycc/Functions/Ptv.h>
#include <pycc/Functions/Ptv_tools.h>
#include <pycc/Functions/Dof_Ptv.h>
#include <pycc/Functions/OrderedPtvSet.h>
#include <pycc/Functions/Dof_OrderedPtvSet.h>



namespace pycc
{

/// This class defines the interface for a finite element.

class fe_Lagrange_1_2D: public ufc::finite_element
{
public:

  /// Constructor
  fe_Lagrange_1_2D();
```

```
  /// Destructor
  virtual ~fe_Lagrange_1_2D();

  /// Return a string identifying the finite element
  virtual const char* signature() const;

  /// Return the cell shape
  virtual ufc::shape cell_shape() const;

  /// Return the dimension of the finite element function space
  virtual unsigned int space_dimension() const;

  /// Return the rank of the value space
  virtual unsigned int value_rank() const;

  /// Return the dimension of the value space for axis i
  virtual unsigned int value_dimension(unsigned int i) const;

  /// Evaluate basis function i at given point in cell
  virtual void evaluate_basis(unsigned int i,
                              double* values,
                              const double* coordinates,
                              const ufc::cell& c) const;

  /// Evaluate order n derivatives of basis function i at given point in cell
  virtual void evaluate_basis_derivatives(unsigned int i,
                                          unsigned int n,
                                          double* values,
                                          const double* coordinates,
                                          const ufc::cell& c) const;

  /// Evaluate linear functional for dof i on the function f
  virtual double evaluate_dof(unsigned int i,
                              const ufc::function& f,
                              const ufc::cell& c) const;

  /// Interpolate vertex values from dof values
  virtual void interpolate_vertex_values(double* vertex_values,
                                         const double* dof_values,
                                         const ufc::cell& c) const;

  /// Return the number of sub elements (for a mixed element)
  virtual unsigned int num_sub_elements() const;

  /// Create a new finite element for sub element i (for a mixed element)
  virtual ufc::finite_element* create_sub_element(unsigned int i) const;

};


} // namespace

#endif
```

C.2.2 Source file for linear Lagrange element in 2D

```
//
// This code complies with UFC version 1.0, and is generated with SyFi version 0.4.0.
//
// http://www.fenics.org/syfi/
// http://www.fenics.org/ufc/
//


#include <stdexcept>
#include <math.h>
#include <ufc.h>
#include <pycc/Functions/Ptv.h>
#include <pycc/Functions/Ptv_tools.h>
#include <pycc/Functions/Dof_Ptv.h>
#include <pycc/Functions/OrderedPtvSet.h>
#include <pycc/Functions/Dof_OrderedPtvSet.h>
#include "fe_Lagrange_1_2D.h"


namespace pycc
{


/// Constructor
fe_Lagrange_1_2D::fe_Lagrange_1_2D() : ufc::finite_element()
{

}

/// Destructor
fe_Lagrange_1_2D::~fe_Lagrange_1_2D()
{

}

/// Return a string identifying the finite element
const char* fe_Lagrange_1_2D::signature() const
{
  return "fe_Lagrange_1_2D // generated by SyFi";
}

/// Return the cell shape
ufc::shape fe_Lagrange_1_2D::cell_shape() const
{
  return ufc::triangle;
}

/// Return the dimension of the finite element function space
unsigned int fe_Lagrange_1_2D::space_dimension() const
{
  return 3;
}

/// Return the rank of the value space
```

```
unsigned int fe_Lagrange_1_2D::value_rank() const
{
  return 0;
}

/// Return the dimension of the value space for axis i
unsigned int fe_Lagrange_1_2D::value_dimension(unsigned int i) const
{
  return 1;
}

/// Evaluate basis function i at given point in cell
void fe_Lagrange_1_2D::evaluate_basis(unsigned int i,
                                      double* values,
                                      const double* coordinates,
                                      const ufc::cell& c) const
{
  const double x = coordinates[0];
  const double y = coordinates[1];
  switch(i)
  {
  case 0:
    values[0] = -x-y+1.0;
    break;
  case 1:
    values[0] = x;
    break;
  case 2:
    values[0] = y;
    break;
  }
}

/// Evaluate order n derivatives of basis function i at given point in cell
void fe_Lagrange_1_2D::evaluate_basis_derivatives(unsigned int i,
                                                  unsigned int n,
                                                  double* values,
                                                  const double* coordinates,
                                                  const ufc::cell& c) const
{
    throw std::runtime_error("gen_evaluate_basis_derivatives not implemented yet.");
}

/// Evaluate linear functional for dof i on the function f
double fe_Lagrange_1_2D::evaluate_dof(unsigned int i,
                                      const ufc::function& f,
                                      const ufc::cell& c) const
{
  // coordinates
  double x0 = c.coordinates[0][0]; double y0 = c.coordinates[0][1];
  double x1 = c.coordinates[1][0]; double y1 = c.coordinates[1][1];
  double x2 = c.coordinates[2][0]; double y2 = c.coordinates[2][1];

  // affine map
  double G00 = x1 - x0;
  double G01 = x2 - x0;
```

```
  double G10 = y1 - y0;
  double G11 = y2 - y0;

  double v[1];
  double x[2];
  switch(i)
    {
  case 0:
    x[0] = x0;
    x[1] = y0;
    break;
  case 1:
    x[0] = x0+G00;
    x[1] = G10+y0;
    break;
  case 2:
    x[0] = G01+x0;
    x[1] = y0+G11;
    break;
  }
  f.evaluate(v, x, c);
  return v[i % 1];

}

/// Interpolate vertex values from dof values
void fe_Lagrange_1_2D::interpolate_vertex_values(double* vertex_values,
                                                 const double* dof_values,
                                                 const ufc::cell& c) const
{
  vertex_values[0] = dof_values[0];
  vertex_values[1] = dof_values[1];
  vertex_values[2] = dof_values[2];
}

/// Return the number of sub elements (for a mixed element)
unsigned int fe_Lagrange_1_2D::num_sub_elements() const
{
  return 1;
}

/// Create a new finite element for sub element i (for a mixed element)
ufc::finite_element* fe_Lagrange_1_2D::create_sub_element(unsigned int i) const
{
  return new fe_Lagrange_1_2D();
}


} // namespace
```

C.2.3 Header file for the dofmap

```
//
// This code complies with UFC version 1.0, and is generated with SyFi version 0.4.0.
//
// http://www.fenics.org/syfi/
// http://www.fenics.org/ufc/
//


#ifndef __dof_map_Lagrange_1_2D_H
#define __dof_map_Lagrange_1_2D_H

#include <stdexcept>
#include <math.h>
#include <ufc.h>
#include <pycc/Functions/Ptv.h>
#include <pycc/Functions/Ptv_tools.h>
#include <pycc/Functions/Dof_Ptv.h>
#include <pycc/Functions/OrderedPtvSet.h>
#include <pycc/Functions/Dof_OrderedPtvSet.h>



namespace pycc
{

/// This class defines the interface for a local-to-global mapping of
/// degrees of freedom (dofs).

class dof_map_Lagrange_1_2D: public ufc::dof_map
{  public:
    pycc::Dof_Ptv dof;
    unsigned int num_elements;
    unsigned int * loc2glob;

public:

  /// Constructor
  dof_map_Lagrange_1_2D();

  /// Destructor
  virtual ~dof_map_Lagrange_1_2D();

  /// Return a string identifying the dof map
  virtual const char* signature() const;

  /// Return true iff mesh entities of topological dimension d are needed
  virtual bool needs_mesh_entities(unsigned int d) const;

  /// Initialize dof map for mesh (return true iff init_cell() is needed)
  virtual bool init_mesh(const ufc::mesh& m);

  /// Initialize dof map for given cell
  virtual void init_cell(const ufc::mesh& m,
                         const ufc::cell& c);
```

```
  /// Finish initialization of dof map for cells
  virtual void init_cell_finalize();

  /// Return the dimension of the global finite element function space
  virtual unsigned int global_dimension() const;

  /// Return the dimension of the local finite element function space
  virtual unsigned int local_dimension() const;

  /// Return the number of dofs on each cell facet
  virtual unsigned int num_facet_dofs() const;

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

  /// Tabulate the local-to-local mapping from facet dofs to cell dofs
  virtual void tabulate_facet_dofs(unsigned int* dofs,
                                   unsigned int facet) const;

  /// Tabulate the coordinates of all dofs on a cell
  virtual void tabulate_coordinates(double** coordinates,
                                    const ufc::cell& c) const;

  /// Return the number of sub dof maps (for a mixed element)
  virtual unsigned int num_sub_dof_maps() const;

  /// Create a new dof_map for sub dof map i (for a mixed element)
  virtual ufc::dof_map* create_sub_dof_map(unsigned int i) const;

};


} // namespace

#endif
```

C.2.4 Source file for the dofmap

```
//
// This code complies with UFC version 1.0, and is generated with SyFi version 0.4.0.
//
// http://www.fenics.org/syfi/
// http://www.fenics.org/ufc/
//


#include <stdexcept>
#include <math.h>
#include <ufc.h>
```

```
#include <pycc/Functions/Ptv.h>
#include <pycc/Functions/Ptv_tools.h>
#include <pycc/Functions/Dof_Ptv.h>
#include <pycc/Functions/OrderedPtvSet.h>
#include <pycc/Functions/Dof_OrderedPtvSet.h>
#include "dof_map_Lagrange_1_2D.h"


namespace pycc
{

/// Constructor
dof_map_Lagrange_1_2D::dof_map_Lagrange_1_2D() : ufc::dof_map()
{
  loc2glob = 0;
}

/// Destructor
dof_map_Lagrange_1_2D::~dof_map_Lagrange_1_2D()
{
  if(loc2glob) delete [] loc2glob;
}

/// Return a string identifying the dof map
const char* dof_map_Lagrange_1_2D::signature() const
{
  return "dof_map_Lagrange_1_2D // generated by SyFi";
}

/// Return true iff mesh entities of topological dimension d are needed
bool dof_map_Lagrange_1_2D::needs_mesh_entities(unsigned int d) const
{
  switch(d)
  {
  case 0: return true; // vertices
  case 1: return true; // edges
  case 2: return true; // faces
  case 3: return false; // volumes
  }
  return false; // strange unsupported case or error
}

/// Initialize dof map for mesh (return true iff init_cell() is needed)
bool dof_map_Lagrange_1_2D::init_mesh(const ufc::mesh& m)
{
  int top_dim = 2;
  num_elements = m.num_entities[top_dim];
  return true;
}

/// Initialize dof map for given cell
void dof_map_Lagrange_1_2D::init_cell(const ufc::mesh& m,
                                      const ufc::cell& c)
{
  // coordinates
  double x0 = c.coordinates[0][0]; double y0 = c.coordinates[0][1];
  double x1 = c.coordinates[1][0]; double y1 = c.coordinates[1][1];
```

```
  double x2 = c.coordinates[2][0]; double y2 = c.coordinates[2][1];

  // affine map
  double G00 = x1 - x0;
  double G01 = x2 - x0;

  double G10 = y1 - y0;
  double G11 = y2 - y0;

  int element = c.entity_indices[2][0];

  double dof0[2] = { x0, y0 };
  Ptv pdof0(2, dof0);
  dof.insert_dof(element, 0, pdof0);

  double dof1[2] = { G00+x0, y0+G10 };
  Ptv pdof1(2, dof1);
  dof.insert_dof(element, 1, pdof1);

  double dof2[2] = { x0+G01, G11+y0 };
  Ptv pdof2(2, dof2);
  dof.insert_dof(element, 2, pdof2);
}

/// Finish initialization of dof map for cells
void dof_map_Lagrange_1_2D::init_cell_finalize()
{
  loc2glob = new unsigned int[num_elements * local_dimension()];
  dof.build_loc2dof(num_elements, local_dimension(), reinterpret_cast<int*>(loc2glob));
}

/// Return the dimension of the global finite element function space
unsigned int dof_map_Lagrange_1_2D::global_dimension() const
{
  return dof.size();
}

/// Return the dimension of the local finite element function space
unsigned int dof_map_Lagrange_1_2D::local_dimension() const
{
  return 3;
}

/// Return the number of dofs on each cell facet
unsigned int dof_map_Lagrange_1_2D::num_facet_dofs() const
{
  return 2;
}

/// Tabulate the local-to-global mapping of dofs on a cell
void dof_map_Lagrange_1_2D::tabulate_dofs(unsigned int* dofs,
                                  const ufc::mesh& m,
                                  const ufc::cell& c) const
{
  const unsigned int *from_dofs = loc2glob + (3 * c.entity_indices[2][0]);
  memcpy(dofs, from_dofs, sizeof(unsigned int)*3);
}
```

```
/// Tabulate the local-to-local mapping from facet dofs to cell dofs
void dof_map_Lagrange_1_2D::tabulate_facet_dofs(unsigned int* dofs,
                                                unsigned int facet) const
{
  switch(facet)
  {
  case 0:
    dofs[0] = 1;
    dofs[1] = 2;
    break;
  case 1:
    dofs[0] = 0;
    dofs[1] = 2;
    break;
  case 2:
    dofs[0] = 0;
    dofs[1] = 1;
    break;
  default:
    throw std::runtime_error("Invalid facet number.");
  }
}

/// Tabulate the coordinates of all dofs on a cell
void dof_map_Lagrange_1_2D::tabulate_coordinates(double** coordinates,
                                                 const ufc::cell& c) const
{
  // coordinates
  double x0 = c.coordinates[0][0]; double y0 = c.coordinates[0][1];
  double x1 = c.coordinates[1][0]; double y1 = c.coordinates[1][1];
  double x2 = c.coordinates[2][0]; double y2 = c.coordinates[2][1];

  // affine map
  double G00 = x1 - x0;
  double G01 = x2 - x0;

  double G10 = y1 - y0;
  double G11 = y2 - y0;


  coordinates[0][0] = x0;
  coordinates[0][1] = y0;
  coordinates[1][0] = G00+x0;
  coordinates[1][1] = y0+G10;
  coordinates[2][0] = x0+G01;
  coordinates[2][1] = G11+y0;
}

/// Return the number of sub dof maps (for a mixed element)
unsigned int dof_map_Lagrange_1_2D::num_sub_dof_maps() const
{
  return 1;
}

/// Create a new dof_map for sub dof map i (for a mixed element)
ufc::dof_map* dof_map_Lagrange_1_2D::create_sub_dof_map(unsigned int i) const
```

```
{
  return new dof_map_Lagrange_1_2D();
}



} // namespace
```

C.2.5 Header file for the stiffness matrix form

```
//
// This code complies with UFC version 1.0, and is generated with SyFi version 0.4.0.
//
// http://www.fenics.org/syfi/
// http://www.fenics.org/ufc/
//


#ifndef __form__stiffness_form__Lagrange_1_2D_H
#define __form__stiffness_form__Lagrange_1_2D_H

#include <stdexcept>
#include <math.h>
#include <ufc.h>
#include <pycc/Functions/Ptv.h>
#include <pycc/Functions/Ptv_tools.h>
#include <pycc/Functions/Dof_Ptv.h>
#include <pycc/Functions/OrderedPtvSet.h>
#include <pycc/Functions/Dof_OrderedPtvSet.h>
#include "dof_map_Lagrange_1_2D.h"
#include "fe_Lagrange_1_2D.h"


namespace pycc
{

/// This class defines the interface for the assembly of the global
/// tensor corresponding to a form with r + n arguments, that is, a
/// mapping
///
///     a : V1 x V2 x ... Vr x W1 x W2 x ... x Wn -> R
///
/// with arguments v1, v2, ..., vr, w1, w2, ..., wn. The rank r
/// global tensor A is defined by
///
///     A = a(V1, V2, ..., Vr, w1, w2, ..., wn),
///
/// where each argument Vj represents the application to the
/// sequence of basis functions of Vj and w1, w2, ..., wn are given
/// fixed functions (coefficients).

class form__stiffness_form__Lagrange_1_2D: public ufc::form
```

```
{
public:

  /// Constructor
  form__stiffness_form__Lagrange_1_2D();

  /// Destructor
  virtual ~form__stiffness_form__Lagrange_1_2D();

  /// Return a string identifying the form
  virtual const char* signature() const;

  /// Return the rank of the global tensor (r)
  virtual unsigned int rank() const;

  /// Return the number of coefficients (n)
  virtual unsigned int num_coefficients() const;

  /// Return the number of cell integrals
  virtual unsigned int num_cell_integrals() const;

  /// Return the number of exterior facet integrals
  virtual unsigned int num_exterior_facet_integrals() const;

  /// Return the number of interior facet integrals
  virtual unsigned int num_interior_facet_integrals() const;

  /// Create a new finite element for argument function i
  virtual ufc::finite_element* create_finite_element(unsigned int i) const;

  /// Create a new dof map for argument function i
  virtual ufc::dof_map* create_dof_map(unsigned int i) const;

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

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

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

};


} // namespace

#endif
```

C.2.6 Source file for the stiffness matrix form

```
//
// This code complies with UFC version 1.0, and is generated with SyFi version 0.4.0.
//
// http://www.fenics.org/syfi/
// http://www.fenics.org/ufc/
//


#include <stdexcept>
#include <math.h>
#include <ufc.h>
#include <pycc/Functions/Ptv.h>
#include <pycc/Functions/Ptv_tools.h>
#include <pycc/Functions/Dof_Ptv.h>
#include <pycc/Functions/OrderedPtvSet.h>
#include <pycc/Functions/Dof_OrderedPtvSet.h>
#include "dof_map_Lagrange_1_2D.h"
#include "fe_Lagrange_1_2D.h"
#include "form__stiffness_form__Lagrange_1_2D.h"


namespace pycc
{

/// This class defines the interface for the tabulation of the cell
/// tensor corresponding to the local contribution to a form from
/// the integral over a cell.

class cell_itg__stiffness_form__Lagrange_1_2D: public ufc::cell_integral
{
public:

  /// Constructor
  cell_itg__stiffness_form__Lagrange_1_2D();

  /// Destructor
  virtual ~cell_itg__stiffness_form__Lagrange_1_2D();

  /// 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;

};


/// Constructor
cell_itg__stiffness_form__Lagrange_1_2D::cell_itg__stiffness_form__Lagrange_1_2D()
: ufc::cell_integral()
{

}

/// Destructor
```

```
cell_itg__stiffness_form__Lagrange_1_2D::~cell_itg__stiffness_form__Lagrange_1_2D()
{

}

/// Tabulate the tensor for the contribution from a local cell
void cell_itg__stiffness_form__Lagrange_1_2D::tabulate_tensor(double* A,
                                    const double * const * w,
                                    const ufc::cell& c) const
{
  // coordinates
  double x0 = c.coordinates[0][0]; double y0 = c.coordinates[0][1];
  double x1 = c.coordinates[1][0]; double y1 = c.coordinates[1][1];
  double x2 = c.coordinates[2][0]; double y2 = c.coordinates[2][1];

  // affine map
  double G00 = x1 - x0;
  double G01 = x2 - x0;

  double G10 = y1 - y0;
  double G11 = y2 - y0;

  double detG_tmp = G00*G11-G01*G10;
  double detG = fabs(detG_tmp);

  double GinvT00 =  G11 / detG_tmp;
  double GinvT01 = -G10 / detG_tmp;
  double GinvT10 = -G01 / detG_tmp;
  double GinvT11 =  G00 / detG_tmp;


  memset(A, 0, sizeof(double)*9);

  A[3*0 + 0] = detG*((GinvT01*GinvT01)/2.0+(GinvT11*GinvT11)/2.0+GinvT10*GinvT00
      +GinvT01*GinvT11+(GinvT00*GinvT00)/2.0+(GinvT10*GinvT10)/2.0);
  A[3*0 + 1] = detG*(-(GinvT01*GinvT01)/2.0-GinvT10*GinvT00/2.0
      -GinvT01*GinvT11/2.0-(GinvT00*GinvT00)/2.0);
  A[3*0 + 2] = detG*(-(GinvT11*GinvT11)/2.0-GinvT10*GinvT00/2.0
      -GinvT01*GinvT11/2.0-(GinvT10*GinvT10)/2.0);
  A[3*1 + 0] = detG*(-(GinvT01*GinvT01)/2.0-GinvT10*GinvT00/2.0
      -GinvT01*GinvT11/2.0-(GinvT00*GinvT00)/2.0);
  A[3*1 + 1] = detG*((GinvT01*GinvT01)/2.0+(GinvT00*GinvT00)/2.0);
  A[3*1 + 2] = detG*(GinvT10*GinvT00/2.0+GinvT01*GinvT11/2.0);
  A[3*2 + 0] = detG*(-(GinvT11*GinvT11)/2.0-GinvT10*GinvT00/2.0
      -GinvT01*GinvT11/2.0-(GinvT10*GinvT10)/2.0);
  A[3*2 + 1] = detG*(GinvT10*GinvT00/2.0+GinvT01*GinvT11/2.0);
  A[3*2 + 2] = detG*((GinvT11*GinvT11)/2.0+(GinvT10*GinvT10)/2.0);
}



/// Constructor
form__stiffness_form__Lagrange_1_2D::form__stiffness_form__Lagrange_1_2D() : ufc::form()
{

}
```

```
/// Destructor
form__stiffness_form__Lagrange_1_2D::~form__stiffness_form__Lagrange_1_2D()
{

}

/// Return a string identifying the form
const char* form__stiffness_form__Lagrange_1_2D::signature() const
{
  return "form__stiffness_form__Lagrange_1_2D // generated by SyFi";
}

/// Return the rank of the global tensor (r)
unsigned int form__stiffness_form__Lagrange_1_2D::rank() const
{
  return 2;
}

/// Return the number of coefficients (n)
unsigned int form__stiffness_form__Lagrange_1_2D::num_coefficients() const
{
  return 0;
}

/// Return the number of cell integrals
unsigned int form__stiffness_form__Lagrange_1_2D::num_cell_integrals() const
{
  return 1;
}

/// Return the number of exterior facet integrals
unsigned int form__stiffness_form__Lagrange_1_2D::num_exterior_facet_integrals() const
{
  return 0;
}

/// Return the number of interior facet integrals
unsigned int form__stiffness_form__Lagrange_1_2D::num_interior_facet_integrals() const
{
  return 0;
}

/// Create a new finite element for argument function i
ufc::finite_element* form__stiffness_form__Lagrange_1_2D::
     create_finite_element(unsigned int i) const
{
  switch(i)
  {
  case 0:
    return new fe_Lagrange_1_2D();
  case 1:
    return new fe_Lagrange_1_2D();
  }
  throw std::runtime_error("Invalid index in create_finite_element()");
}

/// Create a new dof map for argument function i
```

```
ufc::dof_map* form__stiffness_form__Lagrange_1_2D::create_dof_map(unsigned int i) const
{
  switch(i)
  {
  case 0:
    return new dof_map_Lagrange_1_2D();
  case 1:
    return new dof_map_Lagrange_1_2D();
  }
  throw std::runtime_error("Invalid index in create_dof_map()");
}

/// Create a new cell integral on sub domain i
ufc::cell_integral* form__stiffness_form__Lagrange_1_2D::
     create_cell_integral(unsigned int i) const
{
  return new cell_itg__stiffness_form__Lagrange_1_2D();
}

/// Create a new exterior facet integral on sub domain i
ufc::exterior_facet_integral* form__stiffness_form__Lagrange_1_2D::
     create_exterior_facet_integral(unsigned int i) const
{
  return 0;
}

/// Create a new interior facet integral on sub domain i
ufc::interior_facet_integral* form__stiffness_form__Lagrange_1_2D::
     create_interior_facet_integral(unsigned int i) const
{
  return 0;
}


} // namespace
```

Appendix D

Python utilities

The UFC distribution includes a set of Python utilities for generating code that conforms to the UFC specification. These utilities consist of format string templates for C++ header files (`.h` files), implementation files (`.cpp`) and combined header and implementation files (`.h` files containing both the declaration and definition of the UFC functions).

The following format strings are provided:

```
function_combined_{header, implementation, combined}
finite_element_{header, implementation, combined}
dof_map_{header, implementation, combined}
cell_integral_{header, implementation, combined}
exterior_facet_integral_{header, implementation, combined}
interior_facet_integral_{header, implementation, combined}
form_{header, implementation, combined}
```

We demonstrate below how to use the format string `form_combined` together with a dictionary that specifies the code to be inserted into the format string. Typically, a form compiler will first generate the code to be inserted into the dictionary and then in a later stage write the generated code to file in UFC format using the provided format strings.

```
from ufc import form_combined

code = {}
code["classname"] = "Poisson",
...
code["rank"] = "    return 2;",
code["num_coefficients"] = "    return 0;",
code["num_cell_integrals"] = "    return 1;",
...

print form_combined % code
```

Appendix E

Installation

The UFC package consists of two parts, the main part being a single header file called `ufc.h`. In addition, a set of Python utilities to simplify the generation of UFC code is provided.

Questions, bug reports and patches concerning the installation should be directed to the UFC mailing list at the address

```
ufc-dev@fenics.org
```

E.1 Installing UFC

To install UFC, simply run

```
scons
sudo scons install
```

This installs the header file ufc.h and a small set of Python utilities (templates) for generating UFC code. Files will be installed under the default prefix.

The installation prefix may be optionally specified, for example

```
scons install prefix=~/local
```

Alternatively, just copy the single header file `src/ufc/ufc.h` into a suitable include directory.

If you do not want to build and install the python extenstion module of UFC, needed by, e.g., PyDOLFIN, you can write

```
sudo enablePyUFC=No
sudo cons install
```

Help with available options and default arguments can be viewed by

```
scons -h
```

Appendix F

UFC versions

To keep things simple, the UFC classes do not have any run time version control. To upgrade to a new UFC version, all libraries and applications must therefore be recompiled with the new header file `ufc.h`.

F.1 Version 1.0

Initial release.

F.2 Version 1.1

The following six functions have been added to the interface:

- `ufc::finite_element::evaluate_dofs`
- `ufc::finite_element::evaluate_basis_all`
- `ufc::finite_element::evaluate_basis_derivatives_all`
- `ufc::dof_map::geometric_dimension`

- `ufc::dof_map::num_entity_dofs`
- `ufc::dof_map::tabulate_entity_dofs`

An implementation of UFC version 1.0 can be recompiled with the header file from UFC version 1.1 without changes to the source code. The new functions introduced in 1.1 will then simply throw an informative exception. (The new functions are virtual but not pure virtual.)

F.3 Version 1.2

The following functions have been modified:

- `ufc::dof_map::local_dimension`

The following functions have been added to the interface:

- `ufc::dof_map::max_local_dimension`

Appendix G

License

The UFC specification, and in particular the header file `ufc.h`, is released into the public domain.

Index