Efficient Representation of Computational Meshes

Anders Logg
Simula Research Laboratory
e–mail: logg@simula.no

Summary We present a simple yet general and efficient approach to representation of computational meshes. Meshes are represented as sets of *mesh entities* of different topological dimensions and their *incidence relations*. We discuss a straightforward and efficient storage scheme for such mesh representations and efficient algorithms for computation of arbitrary incidence relations from a given initial and minimal set of incidence relations. The general representation may harbor a wide range of computational meshes, and may also be specialized to provide simple user interfaces for particular meshes, including simplicial meshes in one, two and three space dimensions where the mesh entities correspond to vertices, edges, faces and cells. Benchmarks are presented to demonstrate efficiency in terms of CPU time and memory usage.

Introduction

The computational mesh is a central component of any software framework for the (mesh-based) solution of partial differential equations. To reduce run-time and enable the solution of large problems, it is therefore important that the computational mesh may be represented efficiently, both in terms of the speed of operations on the mesh or access of mesh data, and in terms of the memory usage for storing any given mesh in memory.

It is furthermore important that the data structure for the representation of the mesh is general enough to harbor a wide range of computational meshes. This generality must also be reflected in the programming interface to the mesh representation, to allow the implementation of general algorithms on the computational mesh. Many algorithms, such as the assembly of a linear system from a finite element variational problem may be implemented similarly for simplicial, quadrilateral and hexahedral meshes if the programming interface to the mesh representation does not enforce a specific interface limited to a specific mesh type. For example, if the entities on the boundary of a mesh (the *facets*) may be accessed in a similar way independently of the mesh dimension and not as *edges* in two space dimensions and *faces* in three space dimensions, one may use the same code to apply boundary conditions in 2D and 3D.

In [9], a very general and flexible representation of computational meshes is presented. The mesh is represented as a *sieve*, which is in general a directed acyclic graph with the mesh entities as points and directed edges describing how the mesh entities are connected. In this paper, we take a slightly less general approach but reuse some of the concepts from [9]. In particular, we will represent the mesh as a set of *mesh entities* (corresponding to the *points* of the sieve) and their *incidence relations*. We also acknowledge the works [4, 5], where similar concepts are defined and where the importance of *mesh iterators* for expressing generic algorithms on computational meshes is advocated.

The data structures and algorithms discussed in this paper are implemented as a C++ library and is distributed as part of DOLFIN, which is freely available from [8]. DOLFIN is a C++ problem-solving environment for ordinary and partial differential equations and is developed as part of the FEniCS project [7, 6] for the automation of computational mathematical modeling. A Python interface to DOLFIN is also available in the form of PyDOLFIN, generated automatically from the C++ library by SWIG [3, 2].

Design goals

When designing the mesh library, we had the following design goals in mind for the mesh representation and its interface. The mesh representation should be *simple*, meaning that the data is represented in terms of basic C++ arrays `unsigned int*` and `double*`; it should be *generic*, meaning that it should not be specialized to say simplicial meshes in one, two and three space dimensions; and it should be *efficient*, meaning that operations on the mesh or access of mesh data should be fast and the storage should require minimal memory usage for any given mesh. Furthermore, the programming interface to the mesh representation should be *intuitive*, meaning that suitable abstractions (classes) should be available, including specialized interfaces for specific types of meshes as well as generic interfaces that enable dimension-independent programming; and it should be *efficient*, meaning that the overhead of the object-oriented interface should be minimized.

Outline

In the following section, we present the basic concepts that define the mesh representation and its interface. We then discuss the data structures of the C++ implementation of the mesh representation in DOLFIN, followed by a discussion of the algorithms used in DOLFIN to compute any given incidence relation from a given minimal set of incidence relations. Next, we demonstrate the programming interface to the mesh library. Finally, we present a series of benchmarks to demonstrate the efficiency of the mesh representation and its implementation followed by some concluding remarks.

Concepts

The mesh representation is based on the following basic concepts: *mesh*, *mesh topology*, *mesh geometry*, *mesh entity* and *mesh connectivity*. Each of these concepts is mapped directly to the corresponding component (class) of the implementation.

A mesh is defined by its topology and its geometry. The mesh topology defines how the mesh is composed of its parts (the mesh entities) and the mesh geometry describes how the mesh is embedded in some metric space, typically $\mathbb{R}^n$ for $n = 1, 2, 3$. A mesh topology (Figure 1) may be specified as a set of mesh entities (the vertices, edges etc.) and their connectivity (incidence relations). Different embeddings (geometries) may be imposed on any given mesh topology to create different meshes, e.g., when moving the vertices of a mesh in an ALE computation. Below, we discuss the two basic concepts mesh entity and mesh connectivity in some detail and also introduce the concept *mesh function*.

Mesh entities

A mesh entity is a pair (d, i), where d is the topological dimension of the mesh entity and where i is a unique index for the mesh entity within its topological dimension, ranging from 0 to $N_d - 1$ with N_d the number of entities of topological dimension d. We let D denote the maximal topological dimension over the mesh entities and set the topological dimension of the mesh equal to D. This is illustrated in Figure 2, where each mesh entity is labeled by its topological dimension and index (d, i).

For convenience, we also name common entities of low topological dimension or codimension. We refer to entities of topological dimension 0 as *vertices*, entities of dimension 1 as *edges*, entities of dimension 2 as *faces*, entities of codimension 1 (dimension $D - 1$) as *facets* and entities of codimension 0 (dimension D) as *cells*. Thus, for a triangular mesh, the edges are also

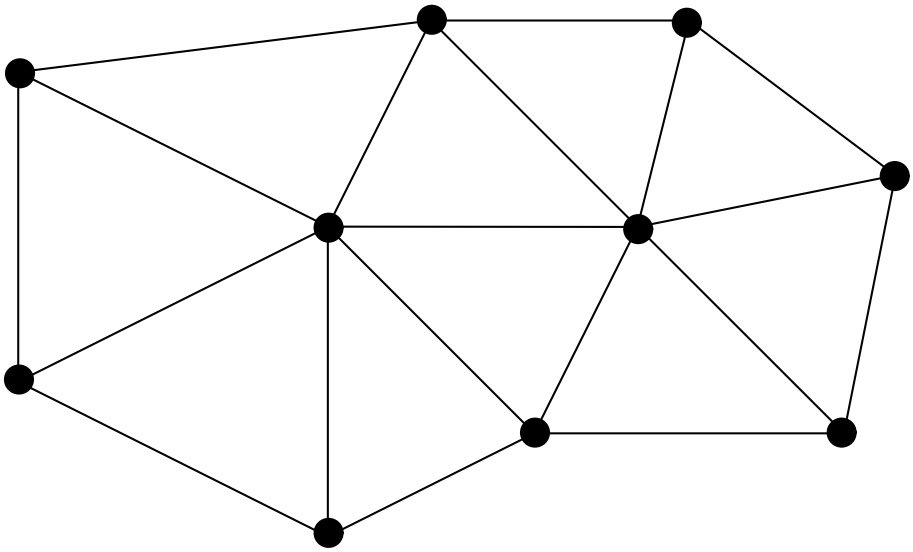

Figure 1: A mesh topology is a set of mesh entities (vertices, edges, etc.) and their connectivity (incidence relations), that is, which entities are connected (incident) to which entities.

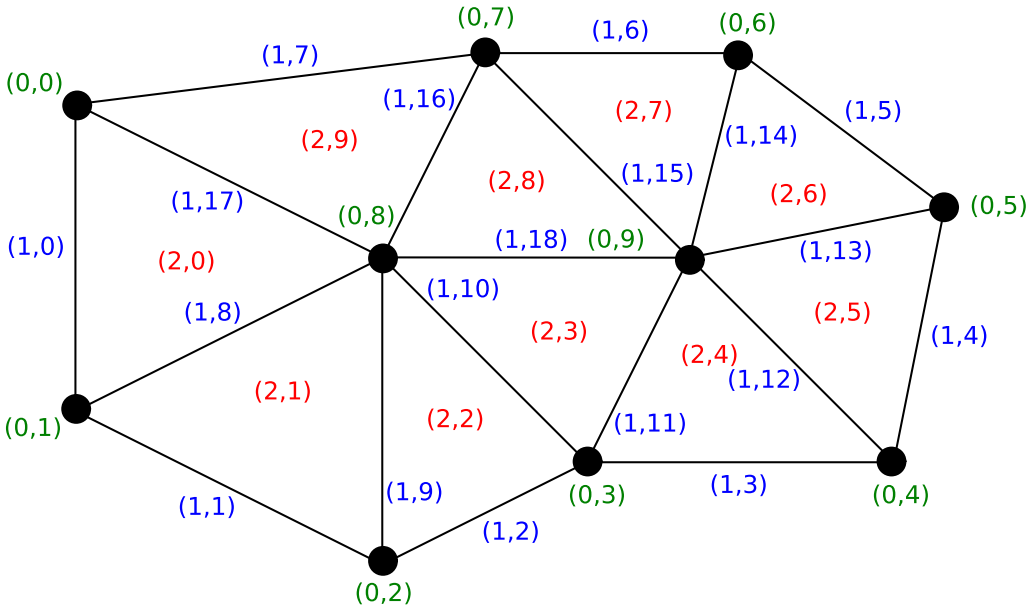

Figure 2: Each mesh entity of a mesh is identified with a pair (d, i), where d is the topological dimension of the mesh entity and where i is a unique index for the mesh entity within its topological dimension, ranging from 0 to $N_d - 1$ with N_d the number of entities of topological dimension d.

facets and the faces are also cells, and for a tetrahedral mesh, the faces are also facets. This is summarized in Table 1.

Entity	Dimension	Codimension
Vertex	0	D
Edge	1	$D-1$
Face	2	$D-2$
Facet	$D-1$	1
Cell	D	0

Table 1: Named entities of low topological dimension or codimension.

Mesh connectivity

We refer to the set of incidence relations on a set of mesh entities as the *connectivity* of the mesh. For a mesh of topological dimension D, there are $(D+1)^2$ different classes of incidence relations (connectivities) to consider. Each such class is denoted here by $d \to d'$ for $0 \leq d, d' \leq D$. For any given mesh entity (d, i), its connectivity $(d \to d')_i$ is given by the set of incident mesh entities of dimension d'.

Thus, for a triangular mesh (of topological dimension $D = 2$), there are nine different incidence relations of interest between the entities of the mesh. These are in turn $0 \to 0$, that is, the vertices incident to each vertex, $0 \to 1$, that is, the edges incident to each vertex, $\ldots$, $D \to D$, that is, the cells incident to each cell.

For $d > d'$, the definition of incidence is evident. Mesh entity (d', i') is incident to mesh entity (d, i) if (d', i') is *contained* in (d, i). Thus, the three vertices of a triangular cell form the set of incident vertices and the three edges form the set of incident edges. For $d < d'$, we define mesh entity (d', i') as incident to mesh entity (d, i) if (d, i) is incident to (d', i'). It thus remains to define incidence for $d = d'$. For $d, d' > 0$, we say that mesh entity (d', i') is incident to mesh entity (d, i) if both are incident to a common vertex, that is, a mesh entity of dimension zero, while for $d = d' = 0$, we say that (d', i') is incident to (d, i) if both are incident to a common cell, that is, a mesh entity of dimension D.

Together, the set of mesh entities and connectivity (incidence relations) define the topology of the mesh. Note that the complete set of incidence relations $d \to d'$ for $0 \leq d, d' \leq D$ may be determined from the single class of incidence relations $D \to 0$, that is, the vertices of each cell in the mesh. We return to this below when we present an algorithm for computing any given class of incidence relations from the minimal set of incidence relations $D \to 0$.

Mesh functions

We define a *mesh function* as a discrete function that takes a value on the set of mesh entities of a given fixed dimension $0 \leq d \leq D$. Mesh functions are simple objects but very useful. A real-valued mesh function may for example be used to describe material parameters on the cells of a mesh. A boolean-valued mesh function may be used to set markers on cells or edges for adaptive refinement. Integer-valued mesh functions may be used to express inter-connectivity between two separate meshes. A typical use is when a boundary mesh is extracted from a given mesh (by identifying the set of facets that are incident to exactly one cell). One may then use a mesh function to describe the mapping from a cell in the extracted boundary mesh (which has topological dimension $D-1$) to the corresponding facets in the original mesh (which has

topological dimension D). Note that mesh functions are discrete and are not meant to represent for example a piecewise polynomial finite element function on the mesh.

Data structures

The mesh representation as described in the previous section has been implemented as a small C++ class library and is available freely as part of the DOLFIN C++ finite element library [8], version 0.6.3 or higher. Each of the basic concepts *mesh*, *mesh topology*, *mesh geometry*, *mesh entity*, *mesh connectivity* and *mesh function* is realized by the corresponding class `Mesh`, `MeshTopology`, `MeshGeometry`, `MeshEntity`, `MeshConnectivity` and `MeshFunction`. All basic data structures are stored as static arrays of unsigned integers (`unsigned int*`) or floating point values (`double*`), which minimizes the cost of storing the mesh data and allows for quick access of mesh data. We discuss each of these classes/data structures in detail below.

The class `Mesh`

The class `Mesh` stores a `MeshTopology` and a `MeshGeometry` that together define the mesh. The `MeshTopology` and `MeshGeometry` are independent of each other and of the `Mesh`. Although it is possible to work with the `MeshTopology` and `MeshGeometry` separately, they are most conveniently accessed through a `Mesh` class that holds a pair of a matching topology and geometry.

The class `MeshTopology`

The class `MeshTopology` stores the topology of a mesh as a set of mesh entities and connectivities. For each pair of topological dimensions $(d, d'), 0 \leq d, d' \leq D$, the class `MeshTopology` stores a `MeshConnectivity` representing the set of incidence relations $d \to d'$. The mesh entities themselves need not be stored explicitly; they are stored implicitly for each topological dimension d as the set of pairs (d, i) for $0 \leq i < N_d$, where N_d is the number of mesh entities of topological dimension d. Thus, for each topological dimension, the class `MeshTopology` stores an (unsigned) integer N_d, from which the set of mesh entities $\{(d, 0), (d, 1), \ldots, (d, N_d - 1)\}$ may be generated.

The class `MeshGeometry`

The class `MeshGeometry` stores the geometry of a mesh. Currently, only the simplest possible representation has been implemented, where only the coordinates of each vertex are stored. These coordinates are stored in a contiguous array `coordinates` of size nN_0, where n is the geometric dimension and N_0 is the number of vertices.

The class `MeshEntity`

The class `MeshEntity` provides a *view* of a given mesh entity (d, i). The mesh entities themselves are not stored, but a `MeshEntity` may be generated from a given pair (d, i). The class `MeshEntity` provides a convenient interface for accessing mesh data, in particular in combination with the concept of mesh iterators, as will be discussed in more detail below. Thus, one may for any given `MeshEntity` access its topological dimension d, its index i and its set of incidence relations (connected mesh entities) of any given topological dimension d'. Specialized interfaces are provided for the named mesh entities of Table 1 in the form of the following sub classes of `MeshEntity`: `Vertex`, `Edge`, `Face`, `Facet` and `Cell`.

The class `MeshConnectivity`

The class `MeshConnectivity` stores the set of incidence relations $d \to d'$ for a fixed pair of topological dimensions (d, d'). The set of incidence relations is stored as a contiguous `unsigned int` array `indices` of entity indices for dimension d' entities, together with an auxiliary `unsigned int` array `offsets` that specifies the offset into the first array for each entity of dimension d.[1] The size of the first array `indices` is equal to the total number of incident entities of dimension d' and the size of the second array `offsets` is equal to the total number of entities of dimension `d` plus one.

As an example, consider the storage of the set of incidence relations $2 \to 0$, that is the vertices of each cell, for the triangular mesh in Figure 3. The mesh has two entities of dimension $d = 2$ and four entities of dimension $d' = 0$. Furthermore, each entity of dimension $d = 2$ is incident to three entities of dimension $d' = 0$. The array `entities` is then given by `[0, 1, 3, 1, 2, 3]` and the array `offsets` is given by `[0, 3, 6]`.

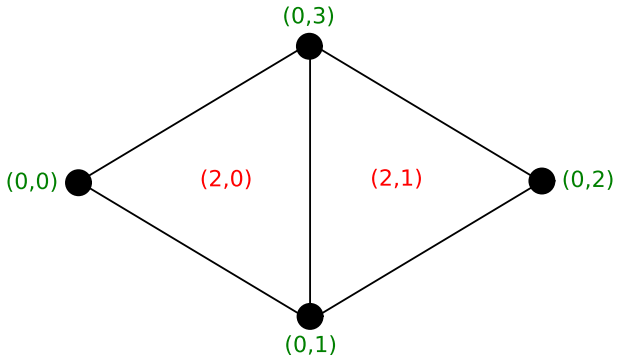

Figure 3: The mesh connectivity $2 \to 0$ (the vertices of each cell) for this triangular mesh with two cells and four vertices is stored as two arrays `indices = [0, 1, 3, 1, 2, 3]` and `offsets = [0, 3, 6]`.

The class `MeshFunction`

The class `MeshFunction` stores a single array of N_d values on the mesh entities of a given fixed dimension d, and is templated over the value type. Typical uses include `MeshFunction<double>` for material parameters that take a constant value on each cell of a mesh, `MeshFunction<bool>` for cell markers that indicate cells that should be refined, and `MeshFunction<unsigned int>` to store inter-mesh connectivity or sub domain markers.

Minimal storage

The mesh data structures described above are summarized in Table 2. We note that the classes `Mesh` and `MeshTopology` function as "aggregate classes" that collect mesh data stored elsewhere, and that no data is stored in the class `MeshEntity`. All data is thus stored in the class `MeshConnectivity` (in the two arrays `indices` and `offsets`) and in the class `MeshGeometry` (in the array `coordinates`). Note that one `MeshConnectivity` object is stored for each pair of topological dimensions (d, d') for which the mesh connectivity has been initialized.

[1] The storage is similar to the standard compressed row storage (CSR) format for sparse matrices, except that only the column indices need to be stored, not the values. Also note that the two arrays `indices` and `offsets` are private data structures of the class `MeshConnectivity`. The user is presented with a more intuitive interface, as will be demonstrated below.

Data structure	Principal data	Size
`Mesh`	`MeshTopology topology` `MeshGeometry geometry`	– –
`MeshTopology`	`MeshConnectivity** connectivities`	–
`MeshGeometry`	`double* coordinates`	nN_0
`MeshEntity`	–	–
`MeshConnectivity`	`unsigned int* indices` `unsigned int* offsets`	$\mathcal{O}(N_d)$ $N_d + 1$

Table 2: Summary of mesh data structures.

As an illustration, consider the storage of a tetrahedral mesh with N_0 vertices and N_3 cells (tetrahedra) embedded in $\mathbb{R}^3$ where we only store the set of incidence relations $D \to 0$. Each cell has four vertices, so the class `MeshConnectivity` stores $4N_3 + N_3 + 1 \sim 5N_3$ values of type `unsigned int`. Furthermore, the class `MeshGeometry` stores $3N_0$ values of type `double`. Thus, if an `unsigned int` is four bytes and a `double` is eight bytes, then the total size of the mesh is $20N_3 + 24N_0$ bytes. For a standard uniform tetrahedral mesh of the unit square, the number of cells is approximately six times the number of vertices, so the total size of the mesh is

$$(20N_3 + 24N_0)\,\mathrm{b} = (20N_3 + 24N_3/6)\,\mathrm{b} = (20N_3 + 4N_3)\,\mathrm{b} = 24N_3\,\mathrm{b}. \tag{1}$$

Thus, a mesh with $1,000,000$ cells may be stored in just 24 Mb. Note that if additional mesh connectivity is computed, like the edges or facets of the tetrahedra, more memory will be required to store the mesh.

Algorithms

In this section, we present the algorithms used by the DOLFIN mesh library to compute the mesh connectivity $d \to d'$ for any given $0 \leq d, d' \leq D$. We assume that we are given an initial set of incidence relations $D \to 0$, that is, we know the vertices of each cell in the mesh.

The key to computing the mesh connectivities of a mesh is to compute the connectivities in a particular order. For example, if the vertices are known for each edge in the mesh ($1 \to 0$), then it is straightforward to compute the edges incident to each vertex ($0 \to 1$) as will be explained below. The computation is based on three algorithms that are used successively in a particular order to compute the desired connectivity. As a consequence, the computation of a certain connectivity $d \to d'$ might require the computation of one or more other connectivities. We describe these algorithms in detail below. An overview is given in Figure 4

Build

Algorithm 1 (Build) computes the connectivities $D \to d$ and $d \to 0$ from $D \to 0$ and $D \to D$ for $0 < d < D$. In other words, given the vertices and incident cells of each cell in the mesh, Algorithm 1 computes the entities of dimension d of each cell and for each such entity the vertices of that entity. Thus, if $d = 1$, then the edges of each cell and the vertices of each edge are computed.

The notation of Algorithm 1 requires some explanation. As before, we let $(d \to d')_i$ denote the set of entities of dimension d' incident to entity (d, i):

$$(d \to d')_i = \{(d', j) : (d', j) \text{ incident to } (d, i)\}. \tag{2}$$

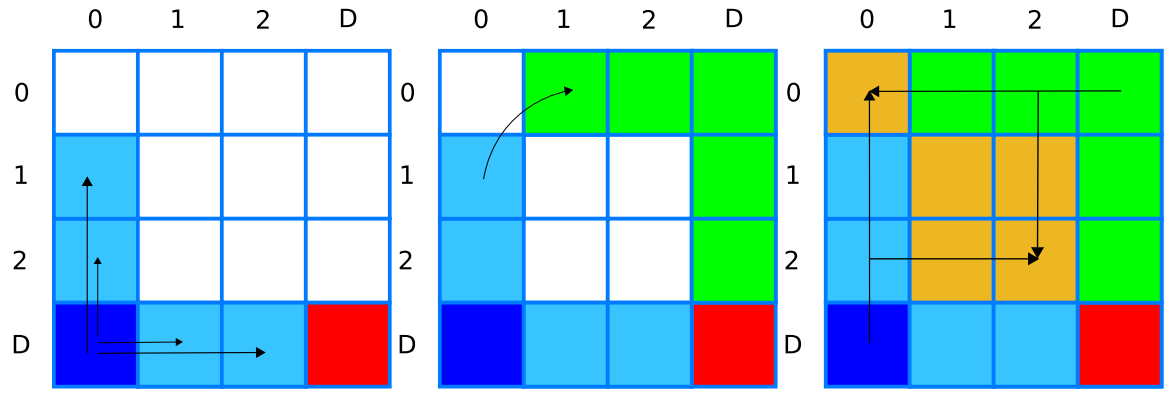

Figure 4: The three basic algorithms for computing connectivity. From the left: Build (computing connectivity $D \to d$ and $d \to 0$ from $D \to 0$ and $D \to D$), Transpose (computing connectivity $d \to d'$ from $d' \to d$) and Intersection (computing connectivity $d \to d'$ from $d \to d''$ and $d'' \to d'$).

Algorithm 1 also uses the operation

$$d \xrightarrow{\text{local}(D,i)} 0, \tag{3}$$

which denotes the set of vertex sets incident to the mesh entities of topological dimension d of a given cell (D, i). To make this concrete, consider a triangular mesh (for which $D = 2$) and take $d = 1$. If $V_i = d \xrightarrow{\text{local}(D,i)} 0$, then V_i denotes the set of vertex sets incident to the edges of triangle number i. The set V_i consists of three sets of vertices (one for each edge) and each set $v_i \in V_i$ contains two vertices. In addition, Algorithm 1 uses the operation

$$\text{index}((D, j), d, v_i), \tag{4}$$

which denotes the index of the entity of dimension d in the cell (D, j) which is incident to the vertices v_i.

We may now summarize Algorithm 1 as follows. For each cell (D, i), we create a set of candidate entities of dimension d, represented by their incident vertices in the set V_i. This operation is local on each cell and must be performed differently for each different type of mesh. We then iterate over each cell incident to the cell (D, i) and check for each candidate entity $v_i \in V_i$ if it has already been created by any of the previously visited cells, making sure that two incident cells agree on the index of any common incident entity.

Transpose

Algorithm 2 (Transpose) computes the connectivity $d \to d'$ from the connectivity $d' \to d$ for $d < d'$. For each entity of dimension d', we iterate over the incident entities of dimension d and add the entities of dimension d' as incident entities to the entities of dimension d. We may thus compute for example the incident cells of each vertex (the cells to which the vertex belongs) by iterating over the cells of the mesh and for each cell over its incident vertices.

Intersection

Algorithm 3 (Intersection) computes the connectivity $d \to d'$ from $d \to d''$ and $d'' \to d'$ for $d \geq d'$. For each entity (d, i) of dimension d, we iterate over each incident entity (d'', k) of dimension d'' and for each such entity we iterate over each incident entity (d', j) of dimension d'. We then check if either (d, i) and (d', j) are entities of the same topological dimension or if

Algorithm 1 Build(d), computing $D \to d$ and $d \to 0$ from $D \to 0$ and $D \to D$ for $0 < d < D$

$k = 0$
for each (D, i)
    $V_i = d \xrightarrow{\text{local}(D,i)} 0$
    for each $(D, j) \in (D \to D)_i$ **such that** $j < i$
        $V_j = d \xrightarrow{\text{local}(D,j)} 0$
        for each $v_i \in V_i$
            if $v_i \in V_j$
                $(D \to d)_i = (D \to d)_i \cup (d, \text{index}((D, j), d, v_i))$
            else
                $(D \to d)_i = (D \to d)_i \cup (d, k)$
                $(d \to 0)_k = v_i$
                $k = k + 1$

Algorithm 2 Transpose(d, d'), computing $d \to d'$ from $d' \to d$ for $d < d'$

for each (d', j)
    for each $(d, i) \in (d' \to d)_j$
        $(d \to d')_i = (d \to d')_i \cup (d', j)$

(d', j) is completely contained in (d, i) by checking that each vertex incident to (d', j) is also incident to (d, i), in which case (d', j') is added as an incident entity of entity (d, i).

Here, d'' must be chosen according to the definition of incidence given above. For example, we may take $d'' = 0$ to compute the connectivity $D \to D$ (the incident cells of each cell) by iterating over the vertices of each cell and for each such vertex iterate over the incident cells.

Algorithm 3 Intersection(d, d', d''), computing $d \to d'$ from $d \to d''$ and $d'' \to d'$ for $d \geq d'$

for each (d, i)
    for each $(d'', k) \in (d \to d'')_i$
        for each $(d', j) \in (d'' \to d')_k$
            if ($d = d'$ **and** $i \neq j$) **or** ($d > d'$ **and** $(d' \to 0)_j \subseteq (d \to 0)_i$)
                $(d \to d')_i = (d \to d')_i \cup (d', j)$

Successive application of Build, Transpose and Intersection

Any given connectivity $d \to d'$ for $0 \leq d, d' \leq D$ may be computed by a successive application of Algorithms 1–3 in a suitable order. In Algorithm 4, we present the basic logic for a successive and recursive application of the three basic algorithms Build, Transpose and Intersection to compute any given connectivity.

We illustrate this in Figure 5 for computation of the connectivity $2 \to 2$, the incident faces of each face, for a tetrahedral mesh. From the given connectivity $D \to 0$, we compute the connectivity $0 \to D$ by an application of Transpose. This allows us to compute $D \to D$ by an application of Intersection. The connectivity $2 \to 0$ (and $D \to 2$) may then be computed by

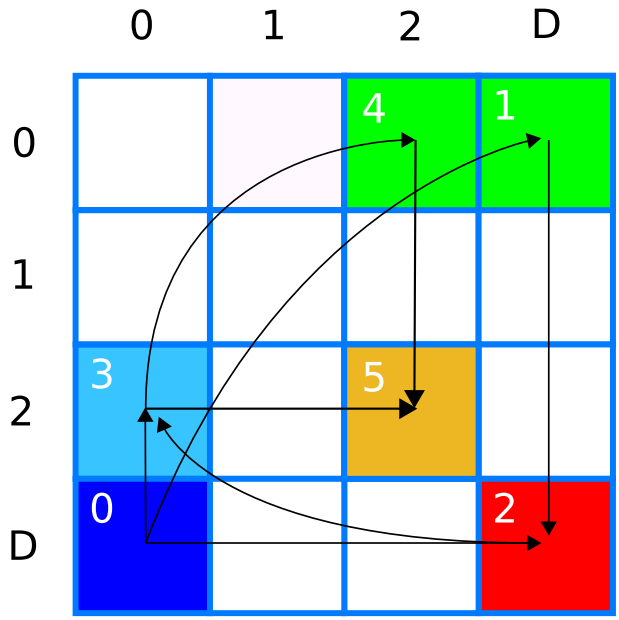

Figure 5: Computing connectivity $2 \rightarrow 2$ (the faces incident to any given face) by successive application of Transpose, Intersection, Build, Transpose and Intersection.

Algorithm 4 Connectivity(d, d'), computing $d \rightarrow d'$ by application of Algorithms 1–3

---

if $N_d = 0$
    Build(d)
if $N_{d'} = 0$
    Build(d')
if $d \rightarrow d' \neq \emptyset$
    return

if $d < d'$
    Connectivity(d', d)
    Transpose(d, d')
else
    if $d = 0$ **and** $d' = 0$
        d'' = D
    else
        d'' = 0
    Connectivity(d, d'')
    Connectivity(d'', d')
    Intersection(d, d', d'')

---

an application of Build. We then apply Transpose to compute $0 \rightarrow 2$ and finally Intersection to compute $2 \rightarrow 2$.

Memory handling

For each of Algorithms 1–3, memory usage may be conserved by running each algorithm twice; first one round to count the number of incident entities, which allows the static data structures discussed above to be preallocated, and then another round to set the values of the incident entities. Furthermore, memory usage may be conserved by clearing incidence relations that get computed as byproducts of Algorithms 1–3 when they are no longer needed.

Interfaces

In this section, we briefly describe the user interface of the DOLFIN mesh library. We only describe the C++ interface, but note that an (almost) identical Python interface is also available.

Creating a mesh

A mesh may be created in one of three ways, as illustrated in Figure 6. Either, the mesh is defined by a data file in the DOLFIN XML format[2], or the mesh is defined vertex by vertex and cell by cell using the DOLFIN mesh editor, or the mesh is defined as one of the DOLFIN built-in meshes. Currently provided built-in meshes include triangular meshes of the unit square and tetrahedral meshes of the unit cube.

```
// Read mesh from file
Mesh mesh0("mesh.xml");

// Build mesh using the mesh editor
Mesh mesh1;
MeshEditor editor;
editor.open(mesh1, "triangle", 2, 2);
editor.initVertices(4);
editor.addVertex(0, 0.0, 0.0);
editor.addVertex(1, 1.0, 0.0);
editor.addVertex(2, 1.0, 1.0);
editor.addVertex(3, 0.0, 1.0);
editor.initCells(2);
editor.addCell(0, 0, 1, 2);
editor.addCell(1, 0, 2, 3);
editor.close();

// Create simple mesh of the unit cube
UnitCube mesh2(16, 16, 16);
```

Figure 6: A DOLFIN mesh may be defined either by an XML data file, or explicitly using the DOLFIN mesh editor, or as a built-in predefined mesh. The last two arguments in the call to `MeshEditor::open()` specify the topological and geometric dimensions of the mesh respectively.

Mesh iterators

Mesh data may be accessed directly from the mesh, but is most conveniently accessed through the mesh iterator interface. Algorithms operating on a mesh (including Algorithms 1–3) can

[2] A conversion script `dolfin-convert` is provided for conversion from other popular mesh formats (including Gmsh and Medit) to DOLFIN XML format.

often be expressed in terms of *iterators*. Mesh iterators can be used to iterate either over the global set of mesh entities of a given topological dimension, or over the locally incident entities of any given mesh entity. Two alternative interfaces are provided; the general interface `MeshEntityIterator` for iteration over entities of some given topological dimension d, and the specialized mesh iterators `VertexIterator`, `EdgeIterator`, `FaceIterator`, `FacetIterator` and `CellIterator` for iteration over named entities. Iteration over mesh entities may be nested at arbitrary depth and the connectivity (incidence relations) required for any given iteration is automatically computed (at the first occurrence) by the algorithms presented in the previous section.

A `MeshEntityIterator` (`it`) may be dereferenced (`*it`) to create a `MeshEntity`, and any member function `MeshEntity::foo()` may be accessed by `it->foo()`. A `MeshEntityIterator` may thus be thought of as a *pointer* to a `MeshEntity`. Similarly, the named mesh entity iterators may be dereferenced to create the corresponding named mesh entities. Thus, dereferencing a `VertexIterator` gives a `Vertex` which provides an interface to access vertex data. For example, if `it` is a `VertexIterator`, then `it->point()` returns the coordinates of the vertex.

The use of mesh iterators is demonstrated in Figure 7, for iteration over all cells in the mesh and for each cell all its vertices as illustrated in Figure 8. For each cell and each vertex, we print its mesh entity index. We also demonstrate the use of named mesh entity iterators to print the coordinates of each vertex.

```
// Iteration over all vertices of all cells
unsigned int D = mesh.topology().dim();
for (MeshEntityIterator cell(mesh, D); !cell.end(); ++cell)
{
  cout << "cell index = " << cell->index() << endl;
  for (MeshEntityIterator vertex(cell, 0); !vertex.end(); ++vertex)
  {
    cout << "vertex index = " << vertex->index() << endl;
  }
}

// Iteration over all vertices of all cells
for (CellIterator cell(mesh); !cell.end(); ++cell)
{
  cout << "cell index = " << cell->index() << endl;
  for (VertexIterator vertex(cell); !vertex.end(); ++vertex)
  {
    cout << "vertex index = " << vertex->index() << endl;
    cout << "vertex coordinates = " << vertex->point() << endl;
  }
}
```

Figure 7: Iteration over all vertices of all cells in a mesh, using the general iterator interface `MeshEntityIterator` and the specialized iterators `CellIterator` and `VertexIterator`.

Direct access to mesh data

In addition to the iterator interface, all mesh data may be accessed directly. Thus, one may obtain an array of the vertices of all cells in the mesh directly from the mesh topology, and one

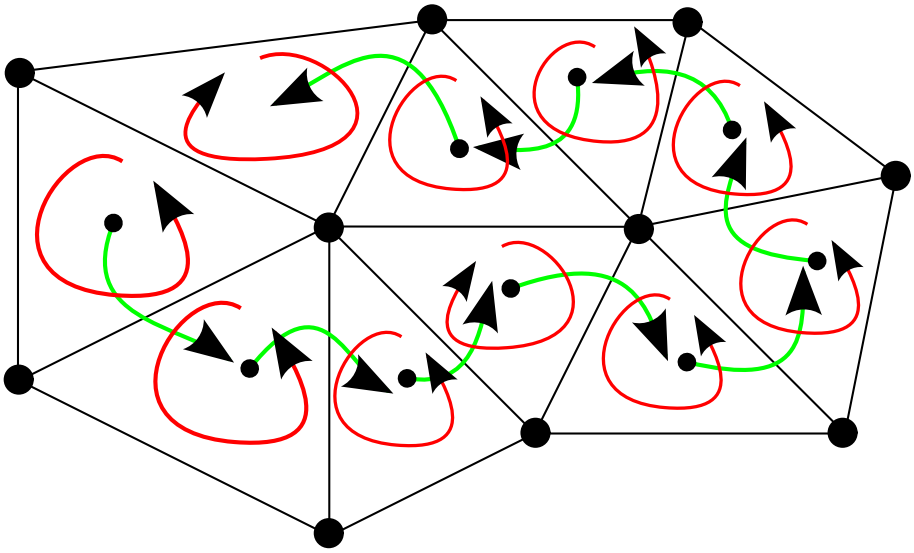

Figure 8: Iteration over all vertices of all cells in a mesh. The order of iteration is decided by the definition of the mesh, or alternatively, the UFC ordering convention [1] if the mesh is ordered. Meshes may be ordered by a call to `Mesh::order()`.

may obtain the vertex coordinates of the mesh directly from the mesh geometry. This illustrated in Figure 9 where the same iteration as in Figure 7 is performed without mesh iterators.

Mesh algorithms

In addition to the computation of mesh connectivity as discussed previously, the DOLFIN mesh library provides a number of other useful mesh algorithms, including boundary extraction, uniform mesh refinement, adaptive mesh refinement (in preparation), mesh smoothing, and reordering of mesh entities.

Figure 10 demonstrates uniform refinement and boundary mesh extraction. When extracting a boundary mesh, it may be desirable to also generate a mapping from the entities of the boundary mesh to the corresponding entities of the original mesh. This is the case for example when assembling the contribution from boundary integrals during assembly of a linear system arising from a finite element variational formulation of a PDE. One then needs to map each cell of the boundary mesh to the corresponding facet of the original mesh. Note that the cells of the boundary mesh are facets of the original mesh. In Figure 11, we demonstrate how to extract a boundary and generate the mapping from the boundary mesh to the original mesh. The mapping is expressed as two `MeshFunction`s, one from the vertices of the boundary mesh to the corresponding vertex indices of the original mesh and one from the cells of the boundary mesh to the corresponding facet indices of the original mesh.

Benchmark results

In this section, we present a series of benchmarks to illustrate the efficiency of the mesh representation and its implementation. The new mesh library, which is available as part of DOLFIN version 0.6.3 and higher, is compared to the old DOLFIN mesh library which is a fairly efficient C++ implementation, but which suffers from object-oriented overhead; all mesh entities are there stored as arrays of objects, which store their data locally in each object (including mesh incidence relations and vertex coordinates).

```
MeshTopology& topology = mesh.topology();
MeshGeometry& geometry = mesh.geometry();
unsigned int D = topology.dim();

for (unsigned int cell = 0; cell < topology.size(D); ++cell)
{
  cout << "cell index = " << cell << endl;

  MeshConnectivity& connectivity = topology(D, 0);
  unsigned int* vertices = connectivity(cell);

  for (unsigned int i = 0; i < connectivity.size(cell); ++i)
  {
    unsigned int vertex = vertices[i];

    cout << "vertex index = " << vertex << endl;
    cout << "vertex coordinates = " << geometry.point(vertex) << endl;
  }
}
```

Figure 9: Iteration over all vertices of all cells in a mesh and direct access of mesh data corresponding to the iteration of Figure 7 and Figure 8.

```
// Refine mesh uniformly twice
mesh.refine();
mesh.refine();

// Extract boundary mesh
BoundaryMesh boundary(mesh);

// Refine boundary mesh uniformly
boundary.refine();

// Save boundary mesh to file
File file("boundary.xml");
file << boundary;
```

Figure 10: Uniform refinement, boundary extraction and uniform refinement of the boundary mesh using the DOLFIN mesh library. Note that the extracted boundary mesh is itself a mesh and may thus itself be refined.

```
MeshFunction<unsigned int> vertex_map;
MeshFunction<unsigned int> cell_map;

BoundaryMesh boundary(mesh, vertex_map, cell_map);
```

Figure 11: Extraction of a boundary mesh and generation of a pair of mappings from the vertices of the boundary mesh to the indices of the corresponding vertices of the original mesh and from the cells of the boundary mesh to the indices of the corresponding facets of the original mesh.

The four test cases that are examined are the following: (i) CPU time and (ii) memory usage for creation of a uniform tetrahedral mesh of the unit cube, (iii) iteration over all vertices of all cells of the uniform tetrahedral mesh of the unit cube, and (iv) uniform refinement of the uniform tetrahedral mesh of the unit cube.

In summary, the speedup was in all cases a factor 10–100 and memory usage was reduced by a factor of 10. The speedup and decreased memory usage is the result of more efficient algorithms and data structures, where all data is stored in large static arrays and objects are only provided as part of the interface for simple access to the underlying data representation, not to store data themselves. Another contributing factor is that the old DOLFIN mesh library precomputes certain connectivities (including the edges and faces of each cell) at startup, whereas this computation is carried out only when requested in the new DOLFIN mesh library, either as part of the iterator interface or by an explicit call to `Mesh::init()`.

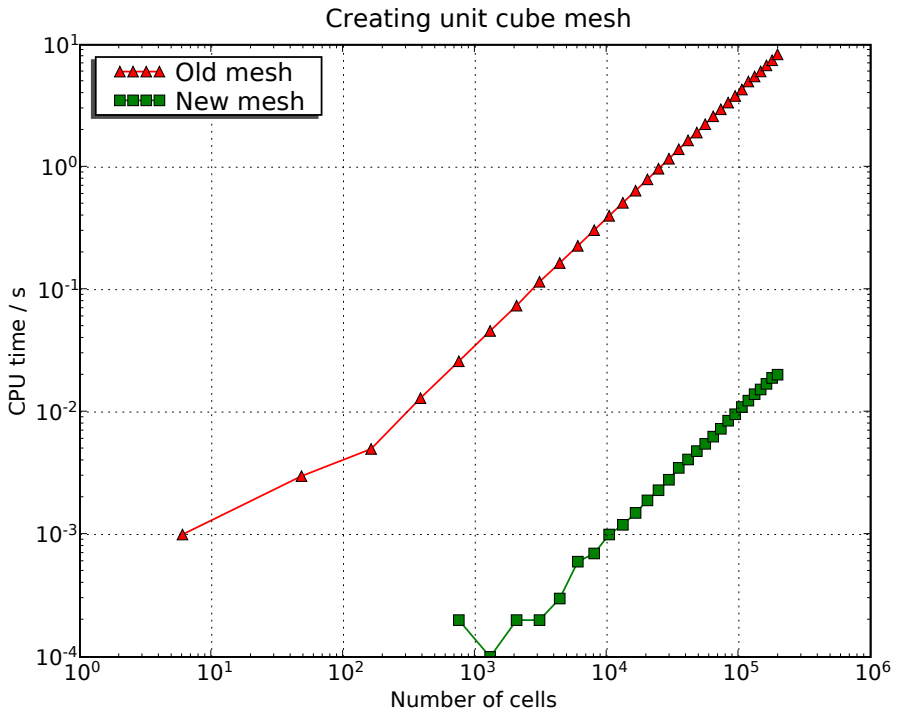

Figure 12: Benchmarking the CPU time for creation of a uniform tetrahedral mesh of the unit cube for the new mesh library vs. the old DOLFIN mesh library.

Conclusions

We have presented a simple yet general and efficient representation of computational meshes and demonstrated a straightforward implementation of this representation as a set of C++ classes that correspond to the basic concepts of the mesh representation. The implementation is available freely as part of DOLFIN [8]. Work is currently in progress to implement adaptive mesh refinement and partition of meshes for parallel computation as part of the DOLFIN mesh library.

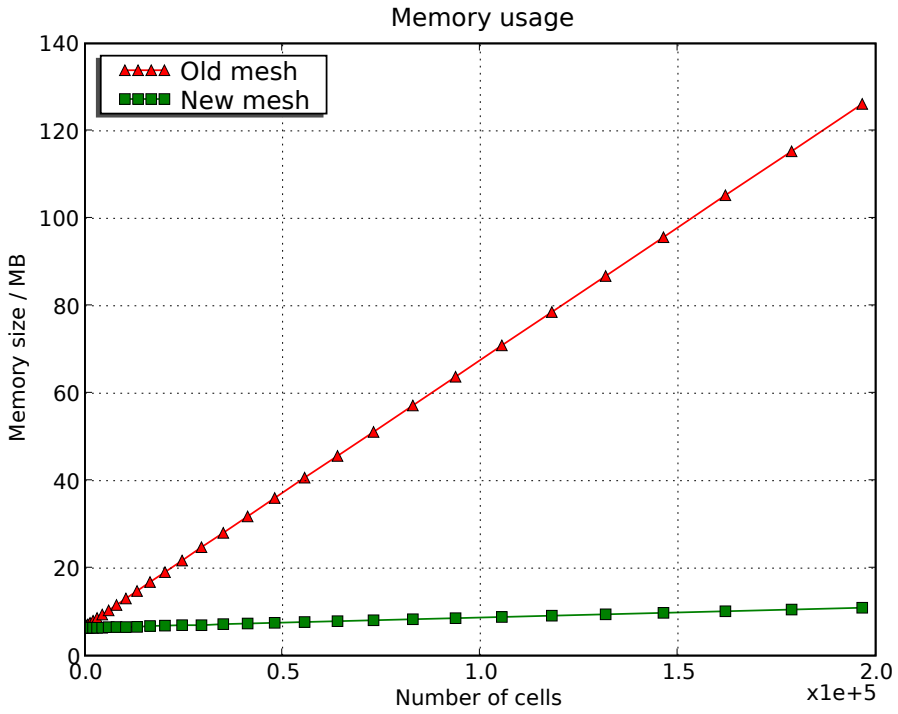

Figure 13: Benchmarking the memory usage for creation of a uniform tetrahedral mesh of the unit cube for the new mesh library vs. the old DOLFIN mesh library.

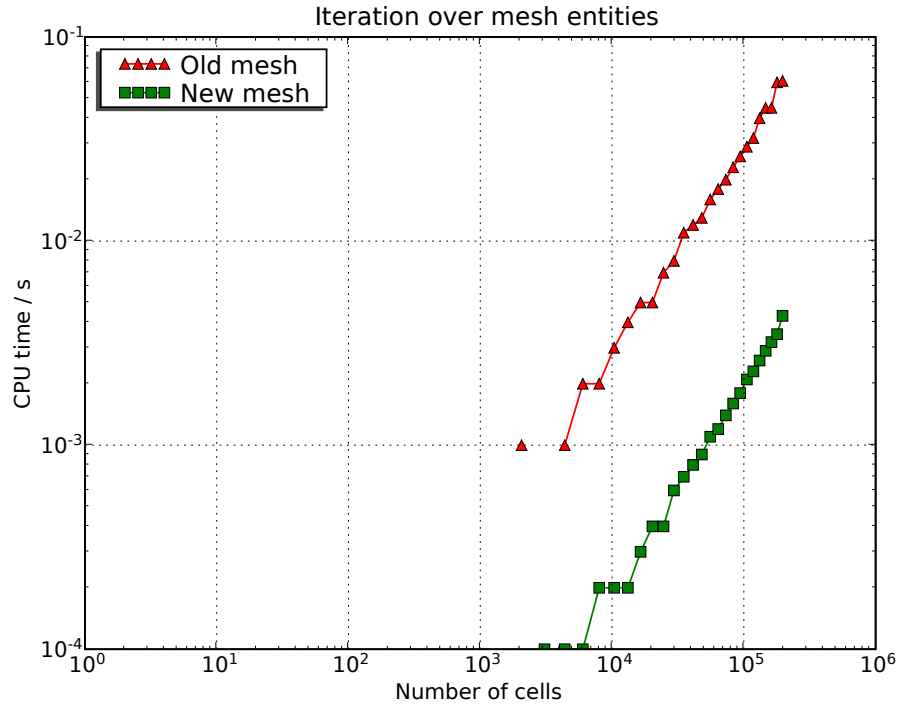

Figure 14: Benchmarking the CPU time for iteration over the vertices of each cell for the new mesh library vs. the old DOLFIN mesh library.

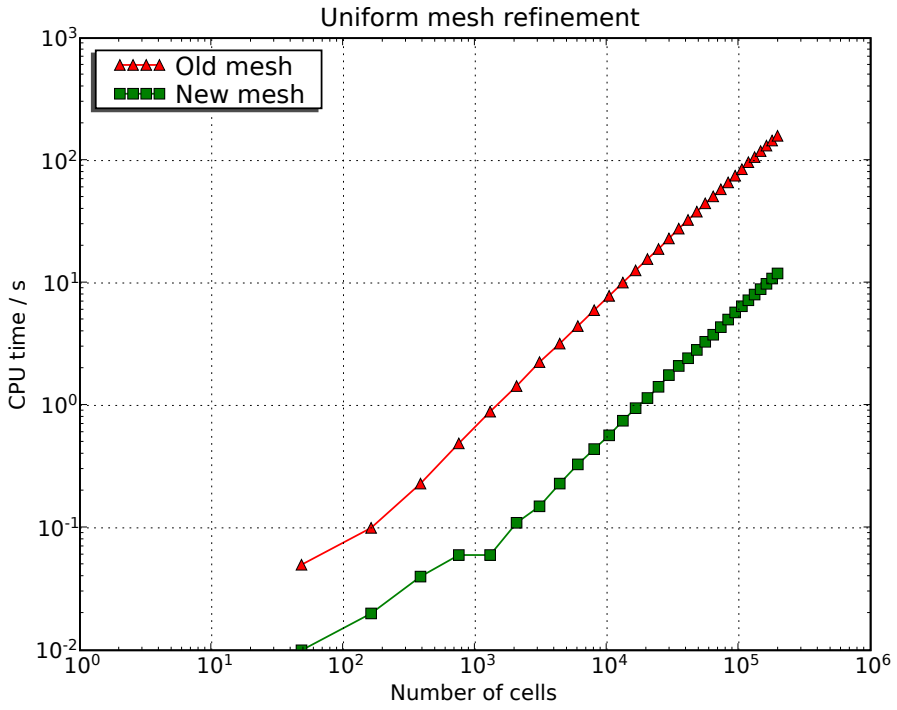

Figure 15: Benchmarking the CPU time for uniform refinement of the unit cube for the new mesh library vs. the old DOLFIN mesh library.

References

[1] M.Alnæs, H.-P.Langtangen, A.Logg, K.-A.Mardal and O.Skavhaug UFC, 2007 URL: `http://www.fenics.org/ufc/`.

[2] D. M.Beazley SWIG : An easy to use tool for integrating scripting languages with C and C++ *presented at the 4th Annual Tcl/Tk Workshop, Monterey, CA*, 2006.

[3] D. M.Beazley et al. Simplified Wrapper and Interface Generator, 2006 URL: `http://www.swig.org/`.

[4] G.Berti Generic programming for mesh algorithms: Towards universally usable geometric components In *Proceedings of the Fifth World Congress on Computational Mechanics,Vienna University of Technology July*, 2002.

[5] G.Berti GrAL - the grid algorithms library *Future Generation Computer Systems*, **vol.22**, 2006.

[6] T.Dupont, J.Hoffman, C.Johnson, R. C.Kirby, M. G.Larson, A.Logg and L. R.Scott The FEniCS project Technical Report 2003–21, Chalmers Finite Element Center Preprint Series, 2003.

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

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

[9] D. A.Karpeev and M. G.Knepley Flexible representation of computational meshes *submitted to ACM Trans. Math. Softw.*, 2005.