Article accepted for publication in Advances in Engineering Software in April 2021

Lightweight finite element mesh database in Julia

Petr Krysl

University of California, San Diego, CA

Abstract

A simple, lightweight, and flexible, package in the programming language Julia for managing finite element mesh data structures is presented. The key role in the design of the data structures is granted to the incidence relation. This concept has some interesting implications for the simplicity and efficiency of the implementation. The entire library has less than 500 executable lines. The low memory requirements are also notable. The design of the data structures is not fixed a priori. The user of the library is given the power over the decisions which mesh entities should be represented explicitly in the data structures, and which of the topological relationships should be computed and stored. This enables a small memory footprint, yet affords a sufficiently rich topology description capability.

Keywords: finite element, mesh, topology, data structure, incidence relation

2010 MSC: 00-01, 99-00

1. Introduction

A number of mesh data structures have been proposed in the literature [1, 2, 3, 4, 5]: radial-edge, winged, half-edge and half-face, entity-based, etc. Usually with the goal of accommodating richer representations of functions on meshes, and supporting complex topological queries. Alas, flexibility and power to support mesh adaptation tend to increase the complexity of the implementation,

Email address: pkrysl@ucsd.edu (Petr Krysl)
URL: http://hogwarts.ucsd.edu/~pkrysl/ (Petr Krysl)

and speed is hard-won in such designs. A common feature of these approaches is the use of pointers to objects in memory [6]. One disadvantage of this implementation is that even when the indices are 32-bit, the pointers on current machines are typically 64-bit. Consequently the memory used for such data structures is not insignificant.

Hence, array-based structures geared towards efficient access, and parsimonious storage of static meshes, also find a receptive ground: STK [7] and MOAB [8, 9] are array-based mesh structures. The innovative and unusual Sieve [10], which in its high-performance incarnation is available as DMPlex [11, 12], is a prime example. Consult also the mesh data structure implemented in FENiCS [13], but that is limited to simplex shapes.

The goal of this paper is to present a simple, lightweight, and flexible, package for managing finite element mesh data structures [14] in the programming language Julia [15, 16]. There are one or two points which the readers may find of interest. The key role is assigned to the incidence relation. This idea has some interesting implications for the simplicity and efficiency of the implementation. The entire library has less than 500 executable lines. The low memory requirements are also of note. Importantly, the data structures are not fixed by the design of the library. The decisions as to (a) which of the of mesh entities of the four manifold dimensions (cells, faces, edges, and vertices) to represent explicitly in the data structures, and(b) which of the 12 topological relationships to compute and store, is configurable on the fly. This is in contrast to the usual "take it or leave it" design.

The database is currently limited to a serial implementation. Adaptivity is also not supported directly, although it can be layered on top of the current design. The data structures are intended for homogeneous meshes, but mixed meshes can be modeled with multiple incidence relations. As such, the presented software is lacking in power compared to some of the full-featured data bases mentioned above. However, this may be (in our opinion, easily) balanced out by the non-negligible advantages the present mesh library has over the heavyweights in terms of flexibility and sparing use of resources. The reader should

keep in mind that the goal is not to replace full-featured mesh databases, but rather to complement them.

Julia is a high-level, dynamic, high-performance, programming language [15, 16]. It is a general-purpose language, but many of its features are closely fitted to the demands of numerical analysis and computational science. Julia's design includes a type system with parametric polymorphism; multiple dispatch as its core programming paradigm; concurrent, parallel and distributed computing (including MPI); direct calling of C and Fortran libraries; agile and efficient compilation approach which uses a "just-ahead-of-time" (JAOT) compiler. The powerful compiler can reason about the Julia code, producing very efficient machine representation, so that well-written programs run at C or Fortran speeds. Yet, at the same time the programming environment provides a read-eval-print loop (REPL) typical of languages such as Python, so that the programmer can engage in interactive work. The language is very consistent and intuitive, so that high productivity is typically reported.

The present library was designed and implemented in Julia, and it is believed that the unique characteristics of this language contributed to the remarkable conciseness and speed of the library. Importantly, we do not use pointers to objects in memory. In fact, we believe that a major factor contributing to the efficiency and simplicity of our library is that it is *not* object-oriented.

The paper is organized as follows: We present the essential ideas and concepts in Section 2, and we describe the basic objects and operations. Section 3 provides some experimental data points concerning the usability, flexibility, and costs of the representation. Discussion and conclusions round off the paper in Section 4.

2. Description of meshes

In finite element analysis there is no such thing as "the" mesh. Even the simplest finite element program will require two meshes: one for the evaluation of the integrals over the interior, and one for the evaluation of the boundary

integrals. Complex finite element programs typically work with a *multitude* of meshes, depending on the requirements of the application. Super-convergent patch recovery, mixed methods, high-order finite element methods with degrees of freedom at the edges, faces, and interiors, in addition to the nodes [4], discontinuous and hybrid Galerkin methods [17], nodal integration methods [18], and so on, need access to mesh entities at various levels of mesh topology. The present mesh library provides enough support for these complex applications, as will be described below.

On the other hand, many basic forms of the finite element method will require only the connectivity, albeit for both the interior and boundary integrals, enumerating for each element its nodes (i.e. a single downward adjacency). If that is so, for efficiency reasons there's no point in constructing and storing additional topological information when it isn't used. Hence, the present library can also attend to the needs of low complexity – low storage requirements cases.

In the next section we describe the basic objects[1] with which the library works: the shape descriptors, and the shape collections, the incidence relations, and the attributes. The reader may also find the Glossary in Appendix A to be of use.

2.1. Shape descriptors, shapes, and shape collections

We consider finite elements here to be ***shapes***, such as line elements, triangles, hexahedra, etc. The shapes are classified according to their manifold dimension, so that we work with the usual vertices (0-dimensional manifolds), line segments (1-dimensional manifolds), triangles and quadrilaterals (2-dimensional manifolds), tetrahedra and hexahedra (3-dimensional manifolds).

The topology of an instance of the shape, which comprises information such as how many nodes are connected together, how many bounding facets there are

[1] We wish to emphasize that we use the term object not in the sense of "object-oriented". The programming language Julia [15, 16] itself is not object-oriented, and our implementation does not attempt graft itself upon the object-oriented tree.

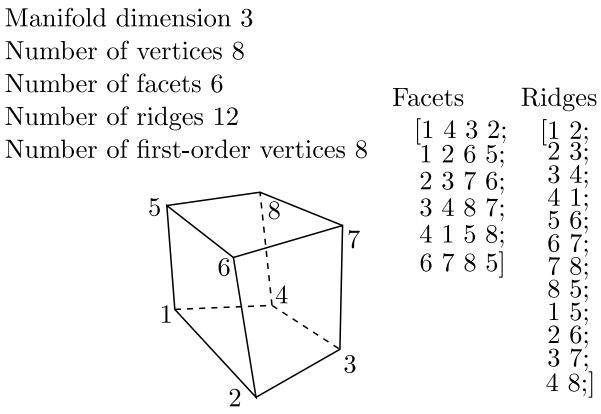

Figure 1: The shape descriptor for an eight-node hexahedron element.

and their definition, is described by ***shape descriptors***. An example of a shape descriptor is provided in Figure 1 which shows the local topological description of a hexahedron shape. The encoding of the topological information into a shape descriptor allows for the functions constructing the incidence relations to work for any shape, no matter what the manifold dimension or order of the element is.

The tables in Figure 1 introduce the concept of facets and ridges [19]: A ***facet*** is a bounding entity: faces for three-dimensional cells, edges for two-dimensional face elements, and vertices for one-dimensional line elements. A ***ridge*** is the "bounding entity of the bounding entity". So edges are the ridges of the three-dimensional cells, and vertices are the ridges of the faces. Edges and vertices have no ridges. A good visual picture of facets and ridges may be the surface of a cut diamond.

The shape itself is not oriented. However, the definition of the facets and ridges in terms of the vertices defines an inherent ***orientation*** (orientability) of the same. Therefore, our algorithms store the orientation of the uses of the facets and ridges in order to facilitate geometric queries. For instance, for the hexahedron the facets are numbered so that when viewed from the outside of the hexahedron, the vertices of each facet are numbered counterclockwise. The ridges are numbered arbitrarily, as there is no intrinsic choice of numbering.

The shapes are considered in the form of collections: ***Shape collections*** are homogeneous collections of shapes. Collections of shapes do not hold any information about how the individual shapes are defined. That is the role of the incidence relations. The shape collections only provide information about the shape descriptor and the attributes of the shape collection, such as geometry (discussed below).

Finally, Figure 1 introduces the so-called ***first-order vertices***. This concept is useful for applications of the library to high-order nodal elements, for instance. As introduced above, when computing relationships between three-dimensional cells and faces or between two dimensional cells (faces) and edges, it is useful to compute the orientation of the uses of the entity. As an example, a quadratic serendipity quadrilateral has eight vertices, but in order to figure out its orientation it is sufficient to refer to its four corner vertices. We call these the first-order vertices: they are the vertices of the first-order versions of the shapes.

2.2. Incidence relation

Our database makes the incidence relation the central idea. This step has also been taken in the Sieve [10] and DMPlex [11]: the "arrows" of the mesh graph. However, we do not think of the mesh in terms of a stratified graph. Our incidence relation is not limited to the representation of the mesh as a bipartite graph.

First, when do we consider entities of the mesh to be ***incident***? An entity E of manifold dimension d_1 is considered to be incident on an entity e of manifold dimension $d_2 \leq d_1$, if e is contained in the ***topological covering*** of the entity E [10]. So, as an example, a tetrahedron is incident on its faces, edges, and vertices. Due to our definition, a tetrahedron is also incident upon itself. The last relation is of use for mixed finite element methods. It provides uniformity of implementation, as is the case for all relations between entities $d_2 = d_1$: an edge, face, or a cell may have degrees of freedom of its own, in which case a map from the mesh entity to other mesh entities will include a map to itself.

Conversely, an entity e of manifold dimension $d_2 \leq d_1$ is incident on an entity E of manifold dimension d_1 if e belongs to E's cover. So a vertex e is incident on all edges, faces, and cells that share it.

By ***incidence relation*** we mean here the relationship between two shape collections. We write

$$(d_L, d_R) \tag{1}$$

where d_L is the manifold dimension of the shape collection on the left of the relation, and d_R is the manifold dimension of the shape collection on the right of the relation. The relationship can be understood as a function which takes as input a serial number of an entity from the shape collection on the *left* and produces as output a list of serial numbers of entities from the shape collection on the *right*, $i_L \rightarrow [j_{R,1}, \ldots j_{R,M}]$. Compare with Table 1 which lists the incidence relations that can be defined unambiguously between entities of the four manifold dimensions. The downward relationships are contained in the lower triangle of the matrix, moving from the bottom of the table upwards, and the upward relationships are listed top to bottom in the upper triangle of the matrix.

The relation $(0, 0)$ between two shape collections that consist of the same set of vertices, possibly in different order, is "trivial": Vertex from the collection on the left is incident on itself in the collection on the right. This mapping may be a permutation, a change of numbering, if necessary or convenient. In general, relations between two shape collections (d, d) are included in Table 1 for reasons relating to mixed finite element methods outlined above: The relation $(0, 0)$ additionally closes the computation of the skeleton (see below).

Computational workflows typically start by creating a collection of d-dimensional shapes, where $d > 0$, such as a tetrahedral mesh produced by a mesh generator, and the collection of shapes is described by the ***connectivity*** (incidence relation) $(d, 0)$. This becomes the starting point for the computation of the required

Table 1: Table of incidence relations. Assuming that the initial mesh is three-dimensional, the first relationship to be established is the connectivity $(3,0)$, as indicated by the box. The surface representation of the boundary, $(2,0)$, would be a derived incidence relation. Other incidence relations may be subsequently computed as discussed in the text. **MD**= Manifold dimension.

MD	**0**	**1**	**2**	**3**
0	$(0,0)$	$(0,1)$	$(0,2)$	$(0,3)$
1	$(1,0)$	$(1,1)$	$(1,2)$	$(1,3)$
2	$(2,0)$	$(2,1)$	$(2,2)$	$(2,3)$
3	$\boxed{(3,0)}$	$(3,1)$	$(3,2)$	$(3,3)$

topological relations, as dictated by the needs of the particular finite element method (refer to Table 1). For definiteness, we assume in the following that we start with a three dimensional mesh (shown boxed in Table 1), so the basic data structure consists of the incidence relation $(3,0)$. Should the initial mesh be two-dimensional, the table would be pruned by removing the fourth row and column.

2.3. Derived Incidence Relations

Here we address the issue of generating any of the other incidence relations of the table on demand. For instance, the incidence relation $(2,0)$ can be derived by application of the skeleton procedure to the incidence relation $(3,0)$. Table 2 lists how the incidence relations in the rows and columns of the table are derived by listing the operation and its arguments. Note the operations are not queries of existing incidence relations. Rather they are incidence relation producers. In Table 2 the only incidence relation that is assumed as given is $(3,0)$, all others are derived using the indicated operations (see examples in Section 2.9).

2.4. `skt`: Skeleton

The incidence relation $(2,0)$ can be derived by application of the procedure "skeleton" (source-code name `ir_skeleton`; here referred to by the abbreviation

Table 2: Operations to populate the table of incidence relations, starting from $(3,0)$. `skt`=skeleton, `trp`=transpose, `bbf`= bounded-by facets, `bbr`= bounded-by ridges, `idt`= identity. **MD**= Manifold dimension.

MD	**0**	**1**	**2**	**3**
0	`skt`$[(1,0)]$	`trp`$[(1,0)]$	`trp`$[(2,0)]$	`trp`$[(3,0)]$
1	`skt`$[(2,0)]$	`idt`$[(1,1)]$	`trp`$[(2,1)]$	`trp`$[(3,1)]$
2	`skt`$[(3,0)]$	`bbf`$[(2,0),(1,0),(0,1)]$	`idt`$[(2,2)]$	`trp`$[(3,2)]$
3	$(3,0)$	`bbr`$[(3,0),(1,0),(0,1)]$	`bbf`$[(3,0),(2,0),(0,2)]$	`idt`$[(3,3)]$

`skt`). Repeated application of the skeleton will yield the relation $(1,0)$, and finally $(0,0)$. Note that at difference to other definitions of the incidence relation $(0,0)$ (the paper of Logg comes to mind [13]) we consider this relation to be one-to-one, not one-to-many.

The skeleton procedure can be implemented in different ways. In our library we use sorting of the connectivity of the entities of the skeleton as a two dimensional array in order to arrive at unique entities by eliminating duplicates (shared entities).

2.5. `bbf`: Bounded-by-facets

The incidence relations $(3,2)$ and $(2,1)$ are obtained by the application of the "bounded-by-facets" procedure (source-code name `ir_bbyfacets`; here referred to by the abbreviation `bbf`). In our implementation the process draws upon three entity relations: the incidence of the mesh entities upon the vertices, and then bidirectional links between the facets and the vertices.

The facets are orientable. Therefore, the incidence relation stores signed entity numbers of the facets: when the facet use traverses the vertices of the facet in the same way in which the facet itself is stored, the orientation is positive (plus sign), and vice versa.

2.6. `bbr`: Bounded-by-ridges

The incidence relation $(3,1)$ is obtained by the application of the "bounded-by-ridges" procedure (source-code name `ir_bbyridges`; here referred to by the

abbreviation `bbr`). The process again draws upon three entity relations: the incidence of the cells upon the vertices, and then bidirectional links between the ridges and the vertices.

The ridges are orientable. Therefore, the incidence relation stores signed entity numbers of the ridges: when the ridge use traverses the vertices of the ridge in the same way in which the ridge itself is stored, the orientation is positive (plus sign), and vice versa.

As an aside, it would also be possible to generate the incidence relation $(2, 0)$ by the "bounded-by-ridges" procedure. It is of course also available by application of the skeleton procedure from the relation $(3, 0)$.

It wouldn't be unreasonable to ask what use is the $(3, 1)$ relation? We can provide at least one instance of clear utility: Take conversion of four-node tetrahedra to the quadratic 10-node tetrahedra. There will be a need to place nodes (vertices) of unique identity at the midpoints of the edges. The conversion routine would therefore build the $(3, 1)$ relation and endow it with an attribute consisting of the newly generated node numbers. The edges being unique will then guarantee the uniqueness of the new nodes. The relation is finally used to construct the connectivity of the new quadratic tetrahedra, that is the $(3, 0)$ relation for the quadratic mesh.

2.7. `trp`: Transpose

All the incidence relations below the diagonal of the matrix of Table 2 yield lists of entities of fixed cardinality. For example, the number of faces, edges, and vertices for hexahedron is always 6, 12, and 8 respectively. On the contrary, the relationships in the upper triangle of the matrix are always of variable cardinality. For example, the number of tetrahedra around an edge [i.e. the incidence relation $(1, 3)$] depends very much upon which edge it is. All the relations above the diagonal are obtained from the relations below the diagonal by the "transpose" operation (source-code name `ir_transpose`; here referred to by the abbreviation `trp`).

2.8. `idt`: Identity

All the incidence relations on the diagonal of the matrix of Table 2 are simple one-to-one (identity) maps (source-code name `ir_identity`; here referred to by the abbreviation `idt`). These operations could also become permutations, if necessary.

This is in contrast to some previously introduced mesh database designs. For instance, Logg [13] defines the incidence relations (d, d), where $d > 1$, as being one-to-many. As an example: the relation $(3, 3)$ in [13] consists of all three-dimensional cells that share a vertex with the cell on the left of the incidence relation. Incidence relations of this type do not fit our definition of incidence of Section 2.2. Even if we were to extend the definition of what we meant by "incident", there would be difficulties: The definition of such a relation would not be unique. In addition to the collections of shapes on the left and on the right, it would have to refer to a third, *connecting*, shape to make sense, which does not fit Table 2 containing relations between two shape collections. For instance, the relationship between faces, $(2, 2)$, would need to state through which shape the incidence occured: is it through a common vertex? Is it through a common edge? Similarly, for cells the incidence relationship $(3, 3)$ would be different for the incidences that follow from a common vertex, from a common edge, or from a common face.

2.9. Constructing the full "one-level" representation

The full "one-level" representation (refer, for example, to [20]), namely the incidence relations downward $(3, 2)$, $(2, 1)$, $(1, 0)$, and upward $(0, 1)$, $(1, 2)$, and $(2, 3)$ can be constructed by our library from the input $(3, 0)$ using the sequence

of operations

$$
\begin{aligned}
(2,0) &= \texttt{skt}[(3,0)] \\
(0,2) &= \texttt{trp}[(2,0)] \\
(3,2) &= \texttt{bbf}[(3,0),(2,0),(0,2)] \\
(1,0) &= \texttt{skt}[(2,0)] \\
(0,1) &= \texttt{trp}[(1,0)] \\
(2,1) &= \texttt{bbf}[(2,0),(1,0),(0,1)] \\
(1,2) &= \texttt{trp}[(2,1)] \\
(2,3) &= \texttt{trp}[(3,2)]
\end{aligned}
\tag{2}
$$

Recall that the arguments in the brackets are *existing* incidence relations. New relations are on the left, and they are returned as output from the operations.

2.10. Relationship to comparable mesh databases

A reasonable question is where does the present database centered on the incidence relation fit in with the concepts described in the Sieve [10] mesh structure, and the DMPlex [11] implementation of these concepts? The DMPlex data structures consist of storing the "points" (the cells, faces, ...), and the "arrows", which are essentially "localized" incidence relations as described here. These data structures are according to the documentation capable of storing the $(d, 0)$ relation or the full "one-level" representation (see Section 2.9).

Consider the diagonal of the matrix in Table 1 as diagonal 0, the diagonal shifted up as +1, and the diagonal below as -1. The full "one-level" representation corresponds to diagonals -1, +1. If we consider the relations on the 0-diagonal as identity, the DMPlex are capable of storing the -1, 0, +1 diagonals of Table 1. When the DMPlex data structures store the full "one-level" representation, the relations on diagonals (+/-)2 and (+/-)3 cannot be stored in the same data structures. In particular, consider that to store simultaneously $(2, 1)$ and $(2, 0)$ would require the two-manifold "points" to store two cones[2] at the

[2] See Knepley and Karpeev for the "Sieve" terminology "point", "cone", etc. [10]: put in a

same time, one cone to 0-manifold points, and one cone to 1-manifold points, which is not allowed by the concept of the cone as a relation between "points" at *two* different heights of the graph. The relations that the DMPlex cannot store can still be calculated on the fly by the user code, but they cannot be interpreted by the database.

We note that it would be possible to construct *additional* mesh databases in addition to the full "one-level" representation to store the incidence relations in the diagonals (+/-)2 and(+/-)3.

Another difference between the present mesh database notions and the Sieve lies in the metadata attached to the stored relations. In the Sieve conceptual structure for the one-level up-down representation, the cones for meshes of four-node tetrahedra and the four-node quadrilaterals all consist of four "points". The DMPlex library does not distinguish between these, as the graph representations are identical. Hence it is up to the user of the library to make sense of the mesh: the interpretation of the mesh is not provided by the database. In contrast, our library maintains the notions of manifold dimension and descriptions of the shape topology, so that the above two meshes are explicitly distinguished.

The incidence relations of Table 1 store in essence the equivalent cone and support information. So we have all the data to form the closure and star queries of the Sieve [10]. Our library is at this point rather minimal, and therefore these queries have not been programmed (yet).

2.11. Mesh

Meshes are understood here simply as incidence relations. At the starting point of a computation, initial meshes are defined by the ***connectivity*** of the finite elements and the finite element nodes, i.e. the incidence relation $(d, 0)$, where $d \geq 0$, linking a d-dimensional shape to a collection of vertices as shapes

simplified way, the cone may be understood as the collection of the mesh entities covering a given entity ("point").

in the form of 0-dimensional manifolds. Any other mesh can be derived by the operations of Table 2.

2.12. Attributes

At a minimum, the geometry of the mesh needs to be defined by specifying the locations of the vertices. In our library we handle this data as attributes of the shape collections. So the locations of the vertices are an attribute of the shape collection of the vertices.

The attribute data can be attached to any incidence relation, which means that associating attributes with edges, faces, cells is easy. For instance, the boundary of a mesh can be derived by calculating the skeleton, which distinguishes then boundary facets from interior facets in an attribute that measures how many times the boundary facet is referenced (boundary facets are referenced just once).

The attribute data is in a sense equivalent to the concept of a "section" from the DMPlex library [11]. In particular, in the DMPlex library the data is attached to the arrows, and here it is attached to the incidence relations.

2.13. Implementation notes

Most mesh databases in current use favor the storage of entity identifiers as 32-bit integers. This allows for substantial ranges of approximately 2 billion positive and 2 billion negative identifiers (which may be useful when storing orientation together with the serial number). If this is not enough, the identifiers may be stored as 64-bit integers. This practically doubles the requisite memory, but considerably expands the range. The present library accommodates storage of the incidence relations with both and either integer types not only in the same library, but also in the same running program: such is the magic of generic programming as implemented in Julia [16] that in the same running program some of the incidence relations may be stored as 32-bit integers while others are stored as 64-bit integers. This mixing is entirely transparent to the user. To get this to work does not require anything beyond specifying parametric types.

As outlined above, the incidence relations below the diagonal differ from the incidence relations above the diagonal by being of fixed cardinality. The implementation in Julia can take proper advantage of this fact while maintaining a single interface to the incidence relations. The incidence relation is stored as a vector of vectors (see 2). Variable-cardinality incidence relations (above the diagonal of the matrix of Table 2) are stored as shown on the left of the figure. This is not as efficient as storing a fixed-cardinality vector of vectors: If all the vectors stored in the master vector are of fixed size, the package `StaticArrays` [21] can be used to enable operations on vectors that can be stored on the stack and that can be in-lined in a vector of vectors as shown in Figure 2. Thus in the fixed-length case, each incidence vector is stored contiguously within one big array (on the right of the figure). Clearly, this saves memory as no storage of pointers for an indirection is needed.

The efficient storage of the fixed-cardinality vector of vectors is enabled by Julia compiler's ability to reason about the code, producing optimized implementation that can take advantage of any information that is known at compile time. At the same time, the programmer sees a uniform interface to the vector of vectors. This is the complete definition of the type of the incidence relation in our library:

```
struct IncRel{LEFT<:AbsShapeDesc, RIGHT<:AbsShapeDesc, T}
  left::ShapeColl{LEFT}   # left shape coll. (L, .)
  right::ShapeColl{RIGHT} # right shape coll. (., R)
  _v::Vector{T}           # vec. of vec.s: shape num.s
  name::String            # name of the inc. relation
end
```

For instance, to find out how many entities in the shape collection on the right are linked to the $j-$th entity in the shape collection on the left we use the definition of the function

```
nentities(ir::IncRel, j) = length(ir._v[j])
```

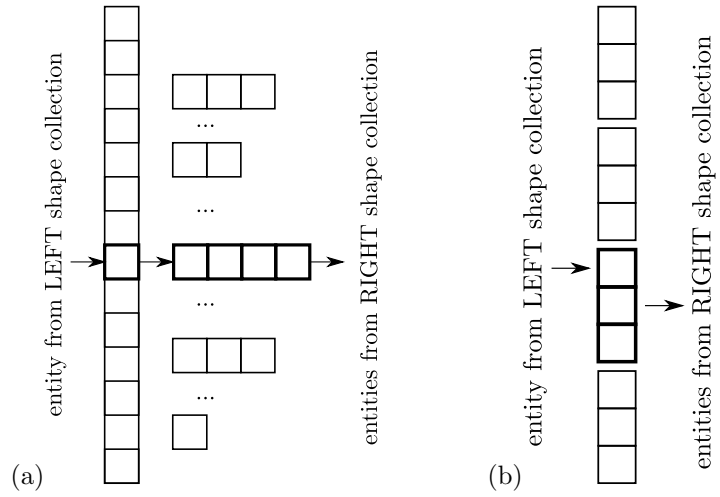

Figure 2: (a) Storage of variable-cardinality vector of vectors on the left. (b) Storage of fixed-cardinality vector of vectors on the right.

Clearly, this function does not distinguish between fixed-cardinality and variable-cardinality vector of vectors.

3. Results

The computations described below were implemented in the Julia programming language [15, 16]. The mesh-topology library is implemented as the `MeshCore.jl` Julia package [14], and the computations referred to in this paper are available to the reader as part of the package `PaperMeshTopo.jl` [22].

3.1. Comparison with MOAB, MDS, GRUMMP, and STK

An interesting comparison of the memory usage for the data structures can be gleaned from Figure 6 of [20]. The mesh is unfortunately not available directly, it is only known that it consists of 100,000 tetrahedra. Hence in the present system we simply generate a tetrahedral mesh of approximately 102,000 elements and compare the resulting storage requirements. Note that we did not generate the measurements for the other data bases ourselves, relying fully on the paper [20]. There is some anecdotal evidence that in the meantime the STK

software changed the way it stores data, and apparently it cannot store flat arrays anymore, and the current memory usage may well be less than reported in [20]. Nevertheless, even with this caveat there is still something that can be learned from this data.

In Figure 3 we compare with the following systems: The MDS database of [20] was the full array representation. MDS-RED referred to as the "reduced array" was the element-to-vertex representation, both using 32-bit indices. Both MDS versions were storing vertex coordinates, geometric model classification, and coordinates of vertices. The MOAB database included element-to-vertex downward and upward adjacency. Apparently only element connectivities and vertices were stored in STK. All of these data bases stored 32-bit indices.

1. First our database was constructed to hold all of the incidence relations that correspond to the *full one-level storage* of MDS. That is we computed and stored the $(3,2)$, $(2,1)$, $(1,0)$ and $(0,1)$, $(1,2)$, and $(2,3)$ incidence relations. "MeshCore 32" refers to this structure with indices stored as 32-bit integers, and "MeshCore 64" refers to the equivalent topology structure with indices stored as 64-bit integers. When we store this information in 32-bit integers, we use only 65% of the memory compared to the MDS.
2. Next, our database was constructed to hold the incidence relations that correspond to the MOAB database with element-to-vertex downward and vertex-to-element upward adjacency. "MeshCore D/U" refers to this structure with indices stored as 32-bit integers. Hence we use only 37% of the storage of MOAB, and 59% of the storage for the MDS-RED.
3. Finally, a third data structure for which we can configure our library, "MeshCore D", stores only the ("down") $(3,0)$ incidence relation. The storage requirement is an order of magnitude smaller than MDS, and amounts to around five times less memory than MOAB. It might not be an appropriate comparison when the storage of more voluminous topological information was justified, but when the finite element program has no use for the additional incidence relations, there's no point in storing

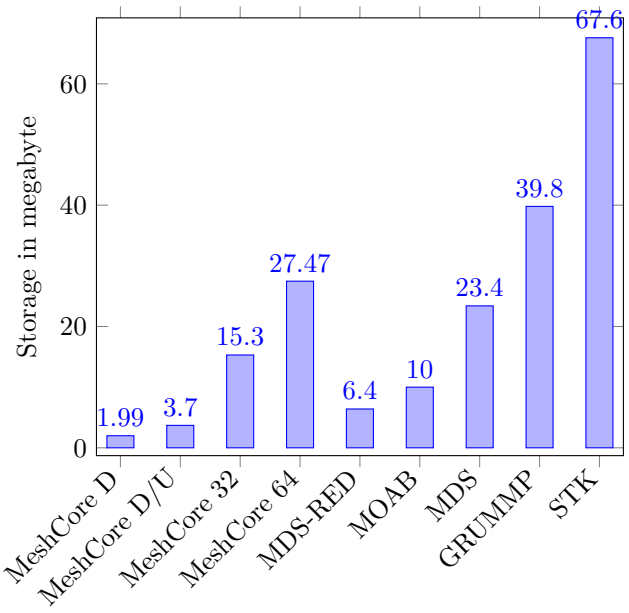

Figure 3: Comparison of the required memory to store various data structures. Legends are discussed in the text.

them, and a mesh storage scheme that avoided this cost would come out ahead. Our design can *freely choose* which incidence relations to store, and therefore we have fine control over the amount of stored information. The comparison data bases commit to the amount of information to store by design. We have the advantage of the structure of the data being configurable according to the needs of the simulation.

We only compare the memory consumption for a non-adaptive use case. We acknowledge that the power of the mesh data bases with which we compare may well justify the increased demand for resources.

3.2. Comparison with DMPlex

Both the present library and DMPlex are centered on the incidence relation (arrows in DMPlex). Here we construct the same mesh of 102,000 four-node tetrahedra in our library and in DMPlex. The memory consumption is recorded

for the 32-bit integer index storage, of the full one-level representation of the mesh. The present library uses 15.3 MB (as reported in Figure 3 as "MeshCore 32"). DMPlex [11] database for this very same mesh uses 22.6 MB of memory.

Of course, we do recognize the greater flexibility of DMPlex: for instance, the incidence relations $(d, d-1)$ are of variable cardinality, whereas ours are of fixed cardinality and hence stored in-line. This flexibility clearly comes with higher storage requirements, in this case 48% higher. It will depend on the application whether or not that is a fair price to pay.

3.3. Performance in finite element analysis

Comparison with DMPlex. On a Windows 10 laptop with an i7 processor and 16 GB of memory, the present library built the initial 100,000-tetrahedron mesh in two seconds and and then created the full one-level representation of the adjacencies in 6.6 seconds. So, the total time was approximately 8.6 seconds.

A PETSC DMPlex sample code was written that built an equivalent tetrahedral mesh and an equivalent one-level mesh database. On the same machine, running within the Windows Subsystem for Linux 2, this code took 10.7 seconds.

In-house mixed finite element package. The presented mesh library is being used, for instance, in the package Elfel.jl [23], which supports mixed finite element methods with basis functions defined at the vertices, edges, faces, and interiors.

A heat conduction problem with 1 million linear quadrilateral elements was completely solved, including mesh generation, calculation of the error, and writing out of postprocessing files, in less than 25 seconds. The assembly of the conductivity matrix and the heat load vector, which is where the access to mesh entities is especially tested, executed in less than 1.3 seconds. This is approximately the same speed as when the this problem was solved with the same mesh in the finite element package deal.II [24].

In three dimensions, the conductivity matrix of a linear-tetrahedron mesh of 3 million elements (approximately 0.5 million degrees of freedom) was assembled in 2.65 seconds.

Next we describe the use of the present library in a mixed finite element method, solving the Stokes problem with the Hood-Taylor approach based on quadratic triangles for the velocities and linear triangles for the pressure. Two meshes were used, sharing the corner nodes. A mesh of 2 million triangles and 9 million degrees of freedom was employed, with an integration rule of three points per triangle, and the saddle-point matrix was constructed in less than 6.5 seconds.

The above calculations were carried out with Julia for Windows 10. In order to judge dependence of the measurements on the operating system, the simulations were also repeated on the same machine with Windows Subsystem for Linux 2 and Julia built for generic Linux. The timings were within 10% of each other. No code needed to be changed to run the model on two different computer architectures.

4. Conclusions

The library `MeshCore.jl` implements a storage model for meshes composed of common shapes such as triangles and quadrilaterals, tetrahedra and hexahedra. All incidence relations (sometimes known as adjacencies) that are commonly encountered in the literature can be produced by the library on the fly, which implements the five operations (skeleton, bounded-by-facets, bounded-by-ridges, transpose, and identity) that can derive for instance the full one-level downward adjacencies (or downward *and* upward adjacencies, if desired). We avoid hardwiring the definition of the topological model in the implementation, at difference to common mesh databases. Our separation of the data model and the implementation allows for a nimble and flexible computation of just the incidence relations that are actually needed. Consequently, the library is very conservative in terms of memory consumption.

Importantly, we also avoid the use of pointers to memory, which is typical with object-oriented mesh databases [6]. Hence we avoid the penalty associated with storing pointers at 64-bits, which is the norm on current computer architectures. Our Julia implementation stores the database in contiguous arrays whenever possible, and transparently switches to vector of vectors for variable-length data.

The current limitations include:

- The data structures do not prevent adaptivity, but the current implementation of the library does not actively support adaptive modifications of the database. It is clear that if the mesh changed, the incidence relations could be recalculated, but not in an incremental fashion.

- Homogeneous meshes are implemented. Mixed-shape meshes (such as a mixture of tetrahedra with a layer of prismatic elements, or a mostly-quadrilateral mesh with some triangles mixed in) can be accommodated by simply storing multiple meshes. Computing incidence relations from multiple meshes requires some care, but does not present major difficulties.

- No consideration has been given at this point to an extension to a distributed database for parallel computations.

The implementation in the Julia language produces code that can be at the same time flexible, powerful, and concise: the entire library has under 500 executable lines, and with copious comments it clocks in at around 1000 lines. This may be contrasted with, for instance, the current version of MOAB which consists of two orders of magnitude larger number of lines of code. Of course, MOAB is much more powerful (it provides import/export, mesh modification, parallel execution). But the point could be made that this creates an opportunity at the other end of the spectrum: something flexible, easy to understand, and small in footprint. We believe that our library fits in that opening quite well.

An interesting opportunity for considerably expanding the usefulness of this

library has been identified by Rypl [25]: due to the generic form of the library, it is suitable for operating on four-dimensional (for instance space-time) meshes. The needed modification entails the addition of a shape descriptor for the four-dimensional cell. The three-dimensional cell would then become a facet, and the faces would become ridges. The tables of incidence relations would acquire a fifth row and column.

Acknowledgments

Continued support by the Office of Naval Research (program manager Michael J. Weise) is gratefully acknowledged. Daniel Rypl of the Czech Technical University in Prague is thanked for numerous thoughtful suggestions.

Author contributions

PK performed the work.

Financial disclosure

None reported.

Conflict of interest

The author declares no potential conflict of interests.

References

[1] L. L. Beghini, A. Pereira, R. Espinha, I. F. M. Menezes, W. Celes, G. H. Paulino, An object-oriented framework for finite element analysis based on a compact topological data structure, Advances in Engineering Software 68 (2014) 40–48. doi:10.1016/j.advengsoft.2013.10.006.

[2] G. Compere, J. F. Remacle, J. Jansson, J. Hoffman, A mesh adaptation framework for dealing with large deforming meshes, International Journal for Numerical Methods in Engineering 82 (7) (2010) 843–867. doi:10.1002/nme.2788.

[3] C. Ollivier-Gooch, L. Diachin, M. S. Shephard, T. Tautges, J. Kraftcheck, V. Leung, X. J. Luo, M. Miller, An interoperable, data-structure-neutral component for mesh query and manipulation, Acm Transactions on Mathematical Software 37 (3). doi:10.1145/1824801.1824807.

[4] E. S. Seol, M. S. Shephard, Efficient distributed mesh data structure for parallel automated adaptive analysis, Engineering with Computers 22 (3-4) (2006) 197–213. doi:10.1007/s00366-006-0048-4.

[5] W. Celes, G. H. Paulino, R. Espinha, A compact adjacency-based topological data structure for finite element mesh representation, International Journal for Numerical Methods in Engineering 64 (11) (2005) 1529–1556. doi:10.1002/nme.1440.

[6] M. W. Beall, M. S. Shephard, A general topology-based mesh data structure, International Journal for Numerical Methods in Engineering 40 (9) (1997) 1573–1596. doi:10.1002/(sici)1097-0207(19970515)40:9<1573::Aid-nme128>3.0.Co;2-9.

[7] H. C. Edwards, A. B. Williams, G. D. Sjaardema, D. G. Baur, W. K. Cochran, Sierra toolkit computational mesh conceptual model, Sandia National Laboratories SAND series, technical report, SAND2010-1192, University of Minnesota, Albuquerque, NM (2010).

[8] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, C. Ernst, MOAB: A mesh-oriented database, Sandia National Laboratories SAND series, technical report, SAND2004-1592, University of Minnesota, Albuquerque, NM (2004).

[9] V. Dyedov, N. Ray, D. Einstein, X. M. Jiao, T. J. Tautges, AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes, Engineering with Computers 31 (3) (2015) 389–404. doi:10.1007/s00366-014-0378-6.

[10] M. G. Knepley, D. A. Karpeev, Mesh algorithms for PDE with Sieve I: Mesh distribution, Scientific Programming 17 (3) (2009) 215–230. doi:10.1155/2009/948613.

[11] M. Lange, L. Mitchell, M. G. Knepley, G. J. Gorman, Efficient mesh management in Firedrake using PETSC DMPLEX, Siam Journal on Scientific Computing 38 (5) (2016) S143–S155. doi:10.1137/15m1026092.

[12] V. Hapla, M. G. Knepley, M. Afanasiev, C. Boehm, M. van Driel, L. Krischer, A. Fichtner, Fully parallel mesh i/o using petsc dmplex with an application to waveform modeling (2020). arXiv:2004.08729.

[13] A. Logg, Efficient representation of computational meshes, International Journal of Computational Science and Engineering 4 (4) (2009) 283–295. doi:10.1504/ijcse.2009.029164.

[14] Petr Krysl, MeshCore: Lightweight Mesh Library in Julia, https://doi.org/10.5281/zenodo.4586905 (Accessed 03/07/2021).

[15] The Julia Project, The Julia programming language, https://julialang.org/ (Accessed 04/11/2019).

[16] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, SIAM review 59 (1) (2017) 65–98.
URL https://doi.org/10.1137/141000671

[17] M. S. Fabien, M. G. Knepley, R. T. Mills, B. M. Riviere, Manycore parallel computing for a hybridizable discontinuous Galerkin nested multigrid method, Siam Journal on Scientific Computing 41 (2) (2019) C73–C96. doi:10.1137/17m1128903.

[18] P. Krysl, B. Zhu, Locking-free continuum displacement finite elements with nodal integration, International Journal for Numerical Methods in Engineering 76 (7) (2008) 1020–1043. doi:10.1002/nme.2354.

[19] J. Matoušek, Lectures on Discrete Geometry, 1st Edition, Graduate Texts in Mathematics, Springer, New York, 2002.

[20] D. Ibanez, M. S. Shephard, Modifiable array data structures for mesh topology, Siam Journal on Scientific Computing 39 (2) (2017) C144–C161. doi:10.1137/16m1063496.

[21] JuliaArrays development team, Statically sized arrays for Julia StaticArrays.jl, https://github.com/JuliaArrays/StaticArrays.jl (Accessed 03/07/2021).

[22] Petr Krysl, PaperMeshTopo: Computations for paper "Lightweight Mesh Library in Julia", https://doi.org/10.5281/zenodo.4587983 (Accessed 03/07/2021).

[23] Petr Krysl, Elfel: Extensible library for finite element methods, https://doi.org/10.5281/zenodo.4587910 (Accessed 03/13/2021).

[24] D. Arndt, W. Bangerth, B. Blais, T. C. Clevenger, M. Fehling, A. V. Grayver, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, R. Rastak, I. Thomas, B. Turcksin, Z. Wang, D. Wells, The deal.II library, version 9.2, Journal of Numerical Mathematics 28 (3) (2020) 131–146. doi:10.1515/jnma-2020-0043.
URL https://dealii.org/deal92-preprint.pdf

[25] D. Rypl, Oral communication (2020).

Appendix A. Glossary

Topological cover: A cover of a set X is a collection of sets whose union contains X as a subset.

Shape: Topological shape of any manifold dimension, 0 for vertices, 1 for edges, 2 for faces, and 3 for cells.

Shape descriptor: Description of the type of the shape, such as the number of vertices, facets, ridges, and so on.

Shape collection: Set of shapes of a particular shape description.

Facet: Shape bounding another shape. A shape is bounded by facets: The facet is a $d-1$ -dimensional face of a d-dimensional entity.

Facet use: Facets are orientable. The incidence relation stores facet uses: when a facet use orders the vertices in the same way (modulo circular shift) as the referenced entity, the facet use is stored as a positive entity number; otherwise it is stored as a negative entity number.

Ridge: Shape one manifold dimension lower than the facet. For instance a tetrahedron is bounded by facets, which in turn are bounded by edges. These edges are the "ridges" of the tetrahedron. The ridges can also be thought of as a "leaky" bounding shapes of 3-D cells. The ridge is a $d-2$ -dimensional face of a d-dimensional entity.

Ridge use: Ridges are orientable. The incidence relation stores ridge uses: when a ridge use orders the vertices in the same way (modulo circular shift) as the referenced entity, the ridge use is stored as a positive entity number; otherwise it is stored as a negative entity number.

Incidence relation: Map from one shape collection to another shape collection. For instance, three-dimensional finite elements (cells) are typically linked to the vertices by the incidence relation $(3, 0)$, i. e. for each tetrahedron the four vertices are listed. Some incidence relations link a shape to a fixed number of other shapes, other incidence relations are of variable cardinality. This is what is usually understood as a "mesh".

Incidence relation operations: The operations defined in the library are: the identity operation, the skeleton operation, the transpose operation, the bounded-by-facets operation, and the bounded-by-ridges operation. All topological relations between the shapes of the four manifold dimensions

that are uniquely defined can be constructed using the sequence of these operations.

Mesh topology: The mesh topology can be understood as an incidence relation between two shape collections.