LIGHTWEIGHT FINITE ELEMENT MESH DATABASE IN JULIA*

PETR KRYSL†

Abstract. A simple, lightweight, and fast, 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 user of the library is given 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.

Key words. finite element, mesh, topology, data structure, incidence relation

AMS subject classifications. 68Q25, 68R10, 68U05

1. Introduction.

A number of mesh data structures have been proposed in the literature [2, 5, 16, 20, 4]: 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, and speed is hard-won in such designs. A common feature of these approaches is the use of pointers to objects in memory [1]. One disadvantage is of course that even when the indexes 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 [21, 6] are array-based mesh structures. An often-cited example is the mesh data structure implemented in FENiCS [14]. It seems also possible to include in this list the innovative and unusual Sieve [11], which in its high-performance incarnation is available as DMPlex [13].

The goal of this paper is to present a simple, lightweight, and fast, package for managing finite element mesh data structures [17] in the programming language Julia [22, 3]. There are one or two points which the readers may find of interest. The key role assigned to the incidence relation appears to be a novel idea, which 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 notice. The present library leaves to the user of the library the decisions on (a) of 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. This is in contrast to the usual "take it or leave it" design. Also, 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 implementation is simple and easy to understand thanks to the Julia programming language [22, 3].

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

*Submitted to the editors DATE.

Funding: This work was not directly supported by a grant or contract.

†University of California, San Diego, CA (pkrysl@ucsd.edu, http://hogwarts.ucsd.edu/~pkrysl/).

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 [20], discontinuous and hybrid Galerkin methods [8], nodal integration methods [12], 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, 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).

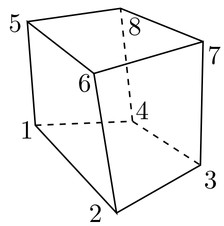
The topology of an instance of the shape, which comprises information such as how many nodes are connected together, how many bounding facets there are and their definition, is described by ***shape descriptors***. An example of a shape descriptor is provided in Figure 1 which shows the local topological 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.

The tables in Figure 1 introduce the concept of facets and ridges [15]: 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 provided by a finely cut diamond on the reader's ring.

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

[1] We wish to emphasize that we use the term object not in the sense of "object-oriented". The programming language Julia [22, 3] itself is not object-oriented, and our implementation does not attempt graft itself upon the object-oriented tree.

Manifold dimension 3
Number of vertices 8
Number of facets 6
Number of ridges 12
Number of first-order vertices 8



Facets

[1 4 3 2;
1 2 6 5;
2 3 7 6;
3 4 8 7;
4 1 5 8;
6 7 8 5]

Ridges

[1 2;
2 3;
3 4;
4 1;
5 6;
6 7;
7 8;
8 5;
1 5;
2 6;
3 7;
4 8;]

FIG. 1. *The shape descriptor for an eight-node hexahedron element.*

numbered so that when viewed from the outside of the hexahedron, each facet vertices 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. 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 cover*** of the entity E. 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, but this last relation is hardly of any use, as is the case for all relations between entities $d_2 = d_1$.

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

$$(d_L, d_R) \tag{2.1}$$

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

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

numbers of entities from the shape collection on the *right*, $i_L \to [j_{R,1}, \dots j_{R,M}]$. 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, change of numbering. It is probably not worthwhile to actually create this relation, but it is included in Table 1 for completeness: it closes the computation of the skeleton (see below). The other relations between two shape collections (d,d) are included to complete the table, but the author is yet to find utility in these incidence relations.

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 topological relations, as dictated by the needs of the particular finite element method (refer to Table 1). For definiteness, in the following we assume 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 the 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. The relations on the diagonal for $d \geq 1$ are omitted, as they result by trivial permutation of the shape collection on the left into the shape collection of the right.

2.4. `skt`: Skeleton. The incidence relation $(2,0)$ can be derived by application of the procedure "skeleton". 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 [14]) 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

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. **MD**= Manifold dimension.*

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

use sorting of the connectivity of the entities of the skeleton as a two dimensional array in order to arrive at unique entities, 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. 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, our 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. 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, our 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 $(3,0)$.

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, 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.

2.8. Constructing the full "one-level" representation. The full "one-level" representation (refer, for example, to [9]), 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

$$
\begin{array}{rcl}
(2,0) & = & \mathtt{skt}[(3,0)] \\
(0,2) & = & \mathtt{trp}[(2,0)] \\
(3,2) & = & \mathtt{bbf}[(3,0),(2,0),(0,2)] \\
(1,0) & = & \mathtt{skt}[(2,0)] \\
(0,1) & = & \mathtt{trp}[(1,0)] \\
(2,1) & = & \mathtt{bbf}[(2,0),(1,0),(0,1)] \\
(1,2) & = & \mathtt{trp}[(2,1)] \\
(2,3) & = & \mathtt{trp}[(3,2)]
\end{array}
\tag{2.2}
$$

2.9. Incidence relations on the diagonal: (d, d)**.** Logg [14] defines the incidence relations (d, d), where $d > 1$, as being one-to-many. For instance, the relation $(3, 3)$ in [14] consists of all three-dimensional cells that are share a vertex with the cell on the left. Such incidence relations do not fit our definition of incidence of Section 2.2. Even if we extend the definition of what we mean by “incident”, there are problems. The definition of such a relation is not unique: In addition to the collections of shapes on the left and on the right, it needs to refer to a connecting shape to make sense, and hence it doesn't fit Table 2. For instance, the relationship between faces, $(2, 2)$, needs to state through which shape the incidence occurs: is it through a common vertex? Is it through a common edge? Similarly, for cells the incidence relationship $(3, 3)$ will be different for the incidences that follow from a common vertex, from a common edge, or from a common face. This is one of the reasons we keep in Table 2 only the incidence relation $(0, 0)$. It fits the definition of incidence, and it is needed as a closure of the skeleton operation.

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

2.10.1. 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.

2.11. 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 [3] 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

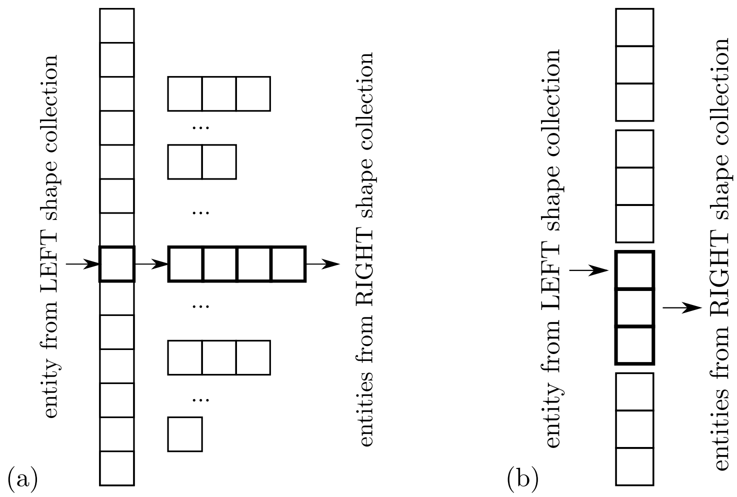


Fig. 2. *(a) Storage of variable-cardinality vector of vectors on the left. (b) Storage of fixed-cardinality vector of vectors on the right.*

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. Variable-cardinality incidence relations (above the diagonal of the matrix of Table 2) are stored as shown on the left. 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` [10] 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. 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 factors 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])
```

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 [22, 3]. The mesh-topology library is implemented as the `MeshCore.jl` Julia package [17], and the computations referred to in this paper are available to the reader as part of the package `PaperMeshTopo.jl` [18].

An interesting comparison of the memory usage for the data structures can be gleaned from Figure 6 of [9]. 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.

In Figure 3 we compare with the following systems: The MDS database of [9] 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.

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 68% of the memory compared to the MDS.

Next, our database was constructed to hold the incidence relations that correspond to the MOAB database with element-to-vertex downward and upward adjacency. "MeshCore D/U" refers to this structure with indices stored as 32-bit integers. Hence we use only 39% of the storage of MOAB, and 61% of the storage for the MDS-RED.

Finally, a third data structure using our library, "MeshCore D", stores only the $(3, 0)$ incidence relation. The storage requirement is an order of magnitude smaller than MDS, and amounts to around five times less memory than MOAB. It may not be an appropriate comparison in situations requiring more voluminous topological information, but if the finite element program has no use for the additional incidence relations, there's no point in storing them, and a mesh storage scheme that can avoid this cost can win big. Our design can freely choose which incidence relations to store, and therefore we have fine control over the amount of stored information. That is the advantage of the structure of the data not being committed to by the design: the amount of information to be stored is left up to the user.

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, which implements the four operations (skeleton, bounded-by-facets, bounded-by-ridges, and transpose) 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.

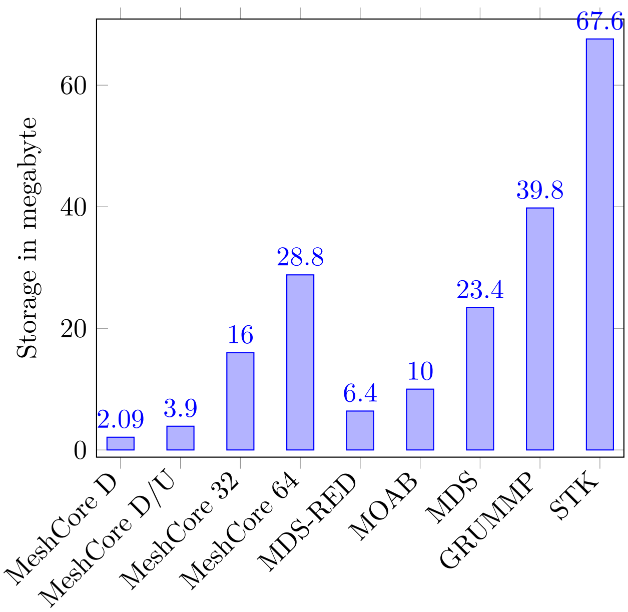


Fig. 3. *Comparison of the required memory to store various data structures. Legends are discussed in the text.*

Importantly, we also avoid the use of pointers to memory, which is typical with object-oriented mesh databases [1]. 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 may allow for adaptivity, but the current implementation of the library is static. At least in the sense that if the mesh changed, the incidence relations could be recalculated, but not in an incremental fashion.
- Homogeneous meshes are implemented. Mixed-shape meshes appear feasible, but have not been implemented yet.
- Support for non-manifold geometries is possible, but so far no effort was expended to reach this goal.
- No consideration has been given at this point to an extension for distributed databases 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 only around 490 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 that leaves open some room 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 [19]: due to the generic form of the library, it is suitable for operating on four-dimensional (for instance time space) 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] M. W. Beall and M. S. Shephard, *A general topology-based mesh data structure*, International Journal for Numerical Methods in Engineering, 40 (1997), pp. 1573–1596, https://doi.org/10.1002/(sici)1097-0207(19970515)40:9⟨1573::Aid-nme128⟩3.0.Co;2-9.

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

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

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

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

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

[7] H. C. Edwards, A. B. Williams, G. D. Sjaardema, D. G. Baur, and 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] M. S. Fabien, M. G. Knepley, R. T. Mills, and B. M. Riviere, *Manycore parallel computing for a hybridizable discontinuous Galerkin nested multigrid method*, Siam Journal on Scientific Computing, 41 (2019), pp. C73–C96, https://doi.org/10.1137/17m1128903.

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

[10] JuliaArrays development team, *Statically sized arrays for Julia StaticArrays.jl*. https://github.com/JuliaArrays/StaticArrays.jl, Accessed 03/25/2020.

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

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

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

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

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

[16] C. Ollivier-Gooch, L. Diachin, M. S. Shephard, T. Tautges, J. Kraftcheck, V. Leung, X. J. Luo, and M. Miller, *An interoperable, data-structure-neutral component for mesh*

query and manipulation, Acm Transactions on Mathematical Software, 37 (2010), https://doi.org/10.1145/1824801.1824807.

[17] PETR KRYSL, *MeshCore: Lightweight Mesh Library in Julia*. https://doi.org/10.5281/zenodo.3737253, Accessed 03/23/2020.

[18] PETR KRYSL, *PaperMeshTopo: Computations for paper "Lightweight Mesh Library in Julia"*. https://github.com/PetrKryslUCSD/PaperMeshTopo.jl, Accessed 03/23/2020.

[19] D. RYPL, *Oral communication*. 2020.

[20] E. S. SEOL AND M. S. SHEPHARD, *Efficient distributed mesh data structure for parallel automated adaptive analysis*, Engineering with Computers, 22 (2006), pp. 197–213, https://doi.org/10.1007/s00366-006-0048-4.

[21] T. J. TAUTGES, R. MEYERS, K. MERKLEY, C. STIMPSON, AND C. ERNST, *MOAB: A mesh-oriented database*, Sandia National Laboratories SAND series, technical report, SAND2004-1592, University of Minnesota, Albuquerque, NM, 2004.

[22] THE JULIA PROJECT, *The Julia programming language*. https://julialang.org/, Accessed 04/11/2019.

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 arity. This is what is usually understood as a "mesh".

Incidence relation operations: The operations defined in the library are: 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.