Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Lightweight finite element mesh database in Julia

Petr Krysl*

[1]Structural Engineering Department, University of California, San Diego, California, USA

Correspondence
*Petr Krysl. Email: pkrysl@ucsd.edu

Summary

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 entrusted to the incidence relation. This novel 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 remarkable. The user of the library is given power over the decisions which mesh entities should be represented explicitly explicitly in the data structures, and which of the topological relationships should be computed and stored. This enables a light memory footprint, yet sufficiently rich topology description capability.

KEYWORDS:
finite element, mesh, topology, data structure

1 | INTRODUCTION

Several designs of mesh data structures for adaptive refinement/unrefinement have been proposed recently: FMDB stores one-level representations in an object-oriented manner[1]. Somewhat similar is the MAdLib of Compère and Remacle[2]. Celes et al are a further example of a mesh data structure suitable for adaptation[3]; see also[4]. Flexibility and power to support mesh adaptation usually leads to complexity and speed is hard-one in such designs.

Hence, array-based structures geared towards efficient access of static meshes also find a receptive ground: STK[5] and MOAB[6,7] are array-based mesh structures. And often-cited example is the mesh data structure implemented in FENiCS[8]. It seems also possible to include in this list the innovative and unusual Sieve[9], which in its high-performance incarnation is available as DMPlex[10].

The goal of this paper is to present a simple, lightweight, and fast, package for managing finite element mesh data structures[11] in the programming language Julia[12,13]. 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 remarkable. The present library leaves 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 13 topological relationships to compute and store, to the user of the library. This is in contrast to the usual "take it or leave it" design.

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 documents some data concerning the usability, flexibility, and costs of the representation. Discussion and conclusions round off the paper in Section 4.

```
Manifold dimension 3
Number of vertices 8
Number of facets 6                          Facets       Edgets
Number of edgets 12                         [1 4 3 2;    [1 2;
Number of first-order vertices 8             1 2 6 5;     2 3;
                                             2 3 7 6;     3 4;
                                             3 4 8 7;     4 1;
                                             4 1 5 8;     5 6;
                                             6 7 8 5]     6 7;
                                                          7 8;
                                                          8 5;
                                                          1 5;
                                                          2 6;
                                                          3 7;
                                                          4 8;]
```

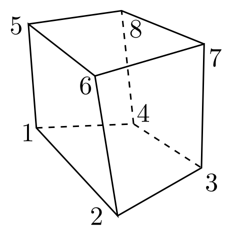

FIGURE 1 The shape descriptor for an eight-node hexahedron element.

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, hi-order finite element methods with degrees of freedom at the edges, faces, and interiors, in addition to the nodes[1], discontinuous and hybrid Galerkin methods[14], nodal integration methods[15], and so on, need access to mesh entities at various levels. 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 (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. 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.

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 with ***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 edgets: A ***facet*** is a bounding entity: faces for three-dimensional cells, edges for two-dimensional face elements, and vertices for one-dimensional line elements. An ***edget*** is the "bounding entity of the bounding entity". So edges are the edgets of the three-dimensional cells, and vertices are the edgets of the faces. Edges and vertices have no edgets.

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).

[1]We wish to emphasize that we use the term object not in the sense of "object-oriented". The programming language Julia[12,13] itself is not object-oriented, and our implementation does not attempt graft itself upon the object-oriented tree.

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 also computed as discussed in the text.

	Manifold dimension			
Manif. dim.	**0**	**1**	**2**	**3**
0	$(0, 0)$	$(0, 1)$	$(0, 2)$	$(0, 3)$
1	$(1, 0)$	–	$(1, 2)$	$(1, 3)$
2	$(2, 0)$	$(2, 1)$	–	$(2, 3$
3	$\boxed{(3, 0)}$	$(3, 1)$	$(3, 2)$	–

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 $d_2 < d_1$ if d_2 is contained in the ***topological cover*** of the entity E. So, as an example, a tetrahedron is incident on its faces, edges, and vertices.

Conversely, an entity e of manifold dimension $d_2 < d_1$ 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*, $i_L \to [j_{R,1}, \dots j_{R,M}]$. Compare with Table 1 which lists the 13 incidence relations that can be defined unambiguously between entities of the four manifold dimensions. The downward relationships in the lower triangle of the matrix, moving from the bottom of the table upwards, and the upward relationships listed top to bottom in the upper triangle of the matrix. The relation (0, 0) is "trivial": Vertex is incident on another vertex (which may be a permutation, change of numbering). Hence it may not be worthwhile to actually create a shape collection for this relation. It is included for completeness, and because it closes the computation of the skeleton (see below).

The collection of d-dimensional shapes is defined by the connectivity $(d, 0)$, where $d > 0$. The starting point for the digesting of the mesh by the computational workflow is typically a two-dimensional or three-dimensional mesh, defined by the connectivity (refer to Table 1). Let us say we start with a three dimensional mesh (shown boxed in Table 1), so the basic data structure consists of the incidence relation (3, 0).

2.3 | Derived Incidence Relations

Table 1 assumes the initial mesh is three-dimensional. If the initial mesh is two-dimensional, the table is pruned by removing the fourth row and column.

Hence, 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 method skeleton 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.

TABLE 2 Operations to populate the table of incidence relations, starting from (3, 0). `sk`=skeleton, `tr`=transpose, `bf`= bounded-by facets, `be`= bounded-by edgets.

	Manifold dimension			
Manif. dim.	**0**	**1**	**2**	**3**
0	sk[(1, 0)]	tr[(1, 0)]	tr[(2, 0)]	tr[(3, 0)]
1	sk[(2, 0)]	–	tr[(2, 1)]	tr[(3, 1)]
2	sk[(3, 0)]	bf[(2, 0), (1, 0), (0, 1)]	–	tr[(3, 2)]
3	(3, 0)	be[(3, 0), (1, 0), (0, 1)]	bf[(3, 0), (2, 0), (0, 2)]	–

2.4 | `sk`: 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[8]) 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, eliminating duplicates (shared) entities.

2.5 | `bf`: 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.

2.6 | `be`: Bounded-by-edgets

The incidence relation (3, 1) is obtained by the application of the “bounded-by-edgets” procedure. The process again draws upon three entity relations: the incidence of the cells upon the vertices, and then bidirectional links between the edgets and the vertices.

As an aside, it would also be possible to generate the incidence relation (2, 0) by the “bounded-by-edgets” procedure. It is of course also available by application of the skeleton procedure.

2.7 | `tr`: Transpose

All the incidence relations below the diagonal of the matrix of Table 2 yield results 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[16]), namely the incidence relations (3, 2), (2, 1), and (1, 0), can be constructed by our library using the sequence of operations

$$\begin{aligned}
(2,0) &= \mathtt{sk}[(3,0)] \\
(0,2) &= \mathtt{tr}[(2,0)] \\
(3,2) &= \mathtt{bf}[(3,0),(2,0),(0,2)] \\
(1,0) &= \mathtt{sk}[(2,0)] \\
(0,1) &= \mathtt{tr}[(1,0)] \\
(2,1) &= \mathtt{bf}[(2,0),(1,0),(0,1)]
\end{aligned} \tag{2}$$

2.9 | 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 $(d, 0)$ linking a d-dimensional manifold, where $d \geq 0$ 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.

2.9.1 | 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 is an attribute of the shape collection of the vertices.

2.10 | 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 both types of integers at the same time: such as the magic of generic programming as implemented in Julia[13] 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 we outlined above, the incidence relations below the diagonal differ from the incidence relations above the diagonal by being of fixed cardinality. The implementation 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. If all the vectors stored in the master vector are of fixed size, the package `StaticArrays`[17] can be used 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: 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 where each incidence vector is stored contiguously within one big array (on the right of the figure). This efficiency is enabled by Julia'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:

```julia
struct IncRel{LEFT<:AbsShapeDesc, RIGHT<:AbsShapeDesc, T}
    left::ShapeColl{LEFT}   # shape collection on the left (L, .)
    right::ShapeColl{RIGHT} # shape collection on the right (., R)
    _v::Vector{T}           # vector of vectors of shape numbers
    name::String            # name of the incidence relation
end
```

3 | RESULTS

The computations described below were implemented in the Julia programming language[12,13]. The library is implemented as the `MeshCore.jl` Julia package[11], and the computation for this paper is 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[16]. 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[16] 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 geo- metric model coordinates at mesh 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.

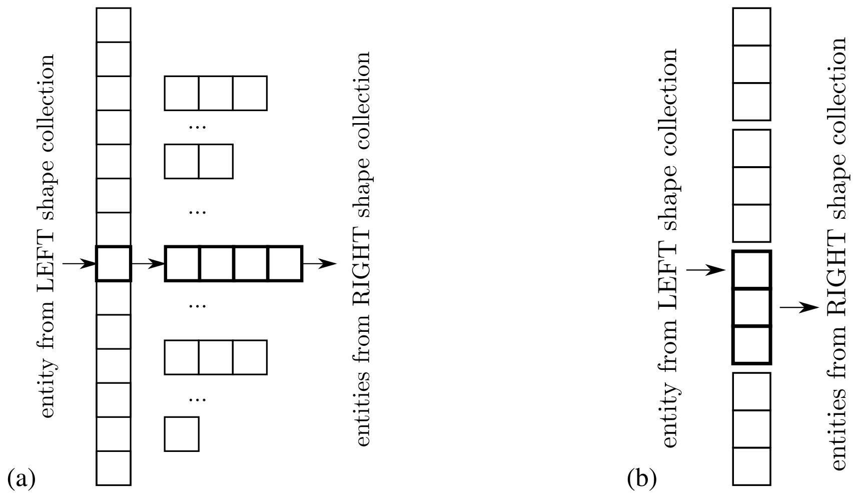
entity from LEFT shape collection
entities from RIGHT shape collection
(a)
entity from LEFT shape collection
entities from RIGHT shape collection
(b)

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

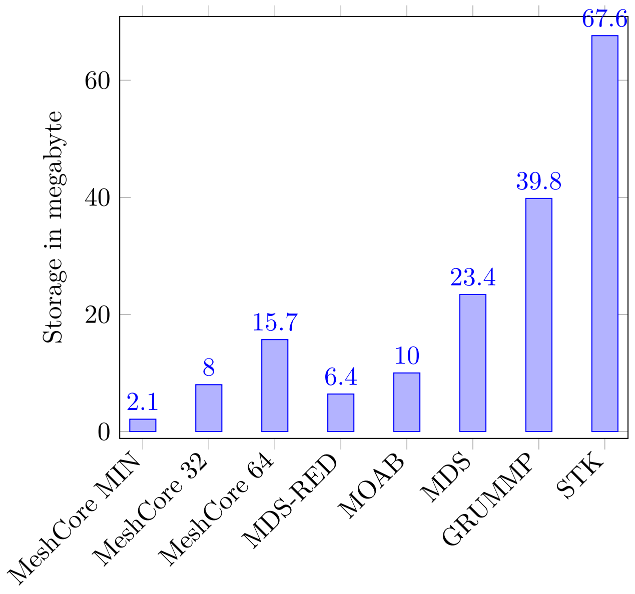

FIGURE 3 This is the sample figure caption.

Our database was constructed to hold all of the incidence relations that correspond to the full one level storage of MDS. That is we compute and store the $(3, 2)$, $(2, 1)$, $(1, 0)$ 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 the topology information equivalent to MDS in 64-bits per index, we only use 75% of the storage required by MDS. When we store this information in 32-bit integers, we use only 34% of the memory. So storing the full topological model we manage to use 20% less storage than MOAB, and 66% less storage than MDS.

It will also be of interest to note a third data structure using our library, "MeshCore MIN", which 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 choose arbitrarily which incidence relations to store, and therefore we have fine control over the amount of stored information.

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-edgets, and transpose) that can produce for instance the full one-level downward adjacencies (or downward and upward adjacencies, if desired). The fact we managed to avoid intertwining the definition of the topological model into the implementation, which seems the common theme in the literature, allows for a nimble and flexible computation of just the incidence relations that are actually needed. Consequently, the library is very light in terms of the memory footprint.

The implementation in the Julia language produces code that can be at the same time concise (the entire library has only 487 executable lines).

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 multiple types of shapes for non-manifold geometries is possible, but no work has been put towards this goal.
- No consideration has been given to an extension for distributed data bases for parallel computations.

ACKNOWLEDGMENTS

This work was partially supported by the Strategic Environmental Research and Development Program (SERDP) Contract No. W912HQ-14-C-0010 and ONR Contract No. N00014-00-D0115 in support of the Effects of Sound on the Marine Environment program. Also, continued support of the first author by the Office of Naval Research (program manager Michael J. Weise) is gratefully acknowledged.

Author contributions

PK performed the work.

Financial disclosure

None reported.

Conflict of interest

The author declares no potential conflict of interests.

References

1. Seol ES, Shephard MS. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers* 2006; 22(3-4): 197-213. doi: 10.1007/s00366-006-0048-4

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

3. Celes W, Paulino GH, Espinha R. A compact adjacency-based topological data structure for finite element mesh representation. *International Journal for Numerical Methods in Engineering* 2005; 64(11): 1529-1556. doi: 10.1002/nme.1440

4. Beghini LL, Pereira A, Espinha R, Menezes IFM, Celes W, Paulino GH. An object-oriented framework for finite element analysis based on a compact topological data structure. *Advances in Engineering Software* 2014; 68: 40-48. doi: 10.1016/j.advengsoft.2013.10.006

5. Edwards HC, Williams AB, Sjaardema GD, Baur DG, Cochran WK. Sierra Toolkit Computational Mesh Conceptual Model. sandia national laboratories sand series, technical report, sand2010-1192, University of Minnesota; Albuquerque, NM: 2010.

6. Tautges TJ, Meyers R, Merkley K, Stimpson C, Ernst C. MOAB: A Mesh-Oriented Database. sandia national laboratories sand series, technical report, sand2004-1592, University of Minnesota; Albuquerque, NM: 2004.

7. Dyedov V, Ray N, Einstein D, Jiao XM, Tautges TJ. AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes. *Engineering with Computers* 2015; 31(3): 389-404. doi: 10.1007/s00366-014-0378-6

8. Logg A. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering* 2009; 4(4): 283-295. doi: 10.1504/ijcse.2009.029164

9. Knepley MG, Karpeev DA. Mesh algorithms for PDE with Sieve I: Mesh distribution. *Scientific Programming* 2009; 17(3): 215-230. doi: 10.1155/2009/948613

10. Lange M, Mitchell L, Knepley MG, Gorman GJ. EFFICIENT MESH MANAGEMENT IN FIREDRAKE USING PETSC DMPLEX. *Siam Journal on Scientific Computing* 2016; 38(5): S143-S155. doi: 10.1137/15m1026092

11. Petr Krysl . MeshCore: Lightweight Mesh Library in Julia. https://github.com/PetrKryslUCSD/MeshCore.jl; Accessed 03/23/2020.

12. The Julia Project . The Julia Programming Language. https://julialang.org/; Accessed 04/11/2019.

13. Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. *SIAM review* 2017; 59(1): 65–98.

14. Fabien MS, Knepley MG, Mills RT, Riviere BM. MANYCORE PARALLEL COMPUTING FOR A HYBRIDIZABLE DISCONTINUOUS GALERKIN NESTED MULTIGRID METHOD. *Siam Journal on Scientific Computing* 2019; 41(2): C73-C96. doi: 10.1137/17m1128903

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

16. Ibanez D, Shephard MS. MODIFIABLE ARRAY DATA STRUCTURES FOR MESH TOPOLOGY. *Siam Journal on Scientific Computing* 2017; 39(2): C144-C161. doi: 10.1137/16m1063496

17. JuliaArrays development team . Statically sized arrays for Julia StaticArrays.jl. https://github.com/JuliaArrays/StaticArrays.jl; Accessed 03/25/2020.

18. Petr Krysl . OmniMesh: Computations for paper Lightweight Mesh Library in Julia. https://github.com/PetrKryslUCSD/PaperMeshTopo.jl; Accessed 03/23/2020.

□

APPENDIX

A GLOSSARY

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

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".

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, edgets, and so on.

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

Facet: shape bounding another shape. A shape is bounded by facets.

Edget: 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 "edgets" of the tetrahedron. The edgets can also be thought of as a "leaky" bounding shapes of 3-D cells.

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

Incidence relation operations: The operations defined in the library are the skeleton, the transpose, the bounded-by for facets, and bounded-by for edgets.