

Building Your Own DEC at Home

Sharif Elcott
Caltech

Peter Schröder
Caltech

1 Overview

The methods of Discrete Exterior Calculus (DEC) have given birth to many new algorithms applicable to areas such as fluid simulation, deformable body simulation, and others. Despite the (possibly intimidating) mathematical theory that went into deriving these algorithms, in the end they lead to simple, elegant, and straightforward implementations. However, readers interested in implementing them should note that the algorithms presume the existence of a suitable simplicial complex data structure. Such a data structure needs to support local traversal of elements, adjacency information for all dimensions of simplices, a notion of a *dual mesh*, and all simplices must be *oriented*. Unfortunately, most publicly available tetrahedral mesh libraries provide only *unoriented* representations with little more than vertex-tet adjacency information (while we need vertex-edge, edge-triangle, edge-tet, *etc.*). For those eager to implement and build on the algorithms presented in this course without having to worry about these details, we provide an implementation of a DEC-friendly tetrahedral mesh data structure in C++. This chapter documents the ideas behind the implementation

1.1 Motivation

Extending a classic pointer-based mesh data structure to 3D is unwieldy, error-prone, and difficult to debug. We instead take a more abstract set-oriented view in the design of our data structure, by turning to the formal definition of an abstract simplicial complex. This gives our implementation the following desirable properties:

- We treat the mesh as a graph and perform all of our operations combinatorially.
- There is no cumbersome pointer-hopping typical of most mesh data structures.
- The design easily generalizes to arbitrary dimension.
- The final result is very compact and simple to implement.

In effect we are taking advantage of the fact that during assembly of all the necessary structures one can use high level, abstract data structures. That way formal definitions can be turned into code almost verbatim. While these data structures (*e.g.*, sets and maps) may not be the most efficient for *computation*, an approach which uses them during assembly is far less error prone. Once everything has been assembled it can be turned easily into more efficient packed representations (*e.g.*, compressed row storage format sparse matrices) with their more favorable performance during the actual computations which occur, *e.g.*, in physical simulation.

1.2 Outline

We will begin with a few definitions in Section 2, and see how these translate into our tuple-based representation in Section 3. The boundary operator, described in Section 4, facilitates mesh traversal and implements the discrete exterior derivative. We show how everything is put together in Section 5. Finally, we discuss our implementation of the DEC operators in Section 6.

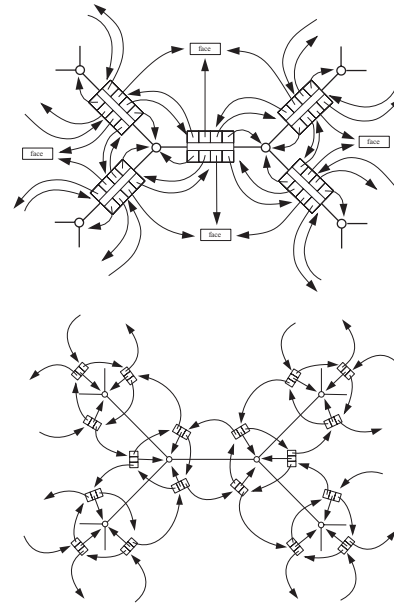


Figure 1: *Some typical examples of 2D mesh representations (from [Joy et al. 2002]; used with permission). Such pointer-based data structures become quite difficult to manage once they are extended to 3D.*

2 Definitions

We begin by recalling the basic definitions of the objects we are dealing with. The focus here is on the rigorous mathematical definitions in a form which then readily translates into high level algorithms. The underlying concepts are simply what we all know informally as *meshes* in either two (triangle) or three (tet) dimensions.

Simplices A *simplex* is a general term for an element of the mesh, identified by its dimension. 0-simplices are vertices, 1-simplices are edges, 2-simplices are triangles, and 3-simplices are tetrahedra.

Abstract Simplicial Complex This structure encodes all the relationships between vertices, edges, triangles, and tets. Since we are only dealing with combinatorics here the atomic element out of which everything is built are the integers $0 \leq i < n$ referencing the underlying vertices. For now they do not yet have point positions in space. Formally, an abstract simplicial complex is a *set of subsets* of the integers $0 \leq i < n$, such that if a subset is contained in the complex then so are all its subsets. For example, a 3D complex is a collection of tetrahedra (4-tuples), triangles (3-tuples), edges (2-tuples), and vertices (singletons), such that if a tetrahedron is present in the complex then so must be its triangles, edges, and vertices. All our simplicial complexes will be proper three or two manifolds, possibly with boundary and may be of arbitrary topology (*e.g.*, containing voids and tunnels).

Manifold The DEC operators that we build on are defined only on meshes which represent manifolds. Practically speaking this means that in a 3D simplicial complex all triangles must have two incident tets only (for a boundary triangle there is only one incident tet). Every edge must have a set of tets incident on it which form a single “ring” which is either open (at the boundary) or closed (in the interior). Finally for vertices it must be true that all incident tets form a topological sphere (or hemisphere at the boundary). These properties should be asserted upon reading the input. For example, for triangles which bound tets one must assert that each such triangle occurs in at most two tets. For an edge the “ring” property of incident tets can be checked as follows. Start with one incident tet and jump across a shared triangle to the next tet incident on the edge. If this walk leads back to the original tet *and* all tets incident on the edge can thusly be visited, the edge passes the test. (For boundary edges such a walk starts at one boundary tet and ends at another.) The test for vertices is more complex. Consider all tets incident on the given vertex. Using the tet/tet adjacency across shared triangles one can build the adjacency graph of all such tets. This graph must be a topological sphere (or hemisphere if the vertex is on the boundary).

Since we need everything to be properly oriented we will only allow *orientable* manifolds (*i.e.*, no Möbius strips or Klein bottles).

Regularity To make life easier on ourselves we also require the simplicial complex to be *strongly regular*. This means that simplices must not have identifications on their boundaries. For example, edges are not allowed to begin and end in the same vertex. Similarly, the edges bounding a triangle must not be identified nor do we allow edges or triangles bounding a tet to be identified. In practice this is rarely an issue since the underlying geometry would need to be quite contorted for this to occur. Strictly speaking though such identifications are possible in more general, abstract settings without violating the manifold property.

Embedding It is often useful to distinguish between the *topology* (neighbor relationships) and the *geometry* (point positions) of the mesh. A great deal of the operations performed on our mesh can be carried out using only topological information, *i.e.*, without regard to the embedding. The embedding of the complex is given by a map $p : [0, n] \mapsto (x, y, z) \in \mathbb{R}^3$ on the vertices (which is extended piecewise linearly to the interior of all simplices). For example, when we visualize a mesh as being composed of piecewise linear triangles (for 2D meshes) or piecewise linear tets, we are dealing with the geometry. Most of the algorithms we describe below do not need to make reference to this embedding. When implementing these algorithms it is useful to only think in terms of combinatorics. There is only one stage where we care about the geometry: the computation of metric dependent quantities needed in the definition of the Hodge star.

3 Simplex Representation

Ignoring orientations for a moment, each k -simplex is represented as a $(k + 1)$ -tuple identifying the vertices that bound the simplex. In this view a tet is simply a 4-tuple of integers, a triangle is a 3-tuple of integers, an edge is a 2-tuple, and a vertex is a singleton. Note that all permutations of a given tuple refer to the same simplex. For example, (i, j, k) and (j, i, k) are different *aliases* for the same triangle. In order to remove ambiguities, we must designate one *representative* alias as the representation of the simplex in our data structures. We do this by using the *sorted* permutation of the

tuple. Thus each simplex (tuple) is stored in our data structures as its canonical (sorted) representative. Then if we, for example, need to check whether two simplices are in fact the same we only need to compare their representatives element by element.

All this information is stored in lists we designate **V**, **E**, **F**, and **T**. They contain one representative for every vertex, edge, triangle, and tet, respectively, in the mesh.

3.1 Forms

The objects of computation in an algorithm using DEC are forms. Formally, a differential k -form is a quantity that can be integrated over a k dimensional domain. For example, consider the expression $\int f(x)dx$ (x being a scalar). The integrand $f(x)dx$ is called a 1-*form*, because it can be integrated over any 1-dimensional interval. Similarly, the dA in $\int \int dA$ would be a 2-form.

Discrete differential forms are dealt with by storing the results of the integrals themselves, instead of the integrands. That is, discrete k -forms associate one value with each k -simplex, representing the integral of the form over that simplex. With this representation we can recover the integral over any k -dimensional chain (the union of some number of k -simplices) by summing the value on each simplex (using the linearity of the integral).

Since all we have to do is to associate one value with each simplex, for our purposes forms are simply vectors of real numbers where the size of the vector is determined by the number of simplices of the appropriate dimension. 0-forms are vectors of size $|\mathbf{V}|$, 1-forms are vectors of size $|\mathbf{E}|$, 2-forms are vectors of size $|\mathbf{F}|$, and 3-forms are vectors of size $|\mathbf{T}|$. Such a vector representation requires that we assign an index to each simplex. We use the position of a simplex in its respective list (**V**, **E**, **F**, or **T**) as its index into the form vectors.

3.2 Orientation

Because the vectors of values we store represent integrals of the associated k -form over the underlying simplices, we must keep track of orientation. For example, reversing the bounds of integration on $\int_a^b f(x)dx$ flips the sign of the resulting value. To manage this we need an *intrinsic orientation* for each simplex. It is with respect to this orientation that the values stored in the form vectors receive the appropriate sign. For example, suppose we have a 1-form f with value f_{ij} assigned to edge $e = (i, j)$; that is, the real number f_{ij} is the integral of the 1-form f over the line segment (p_i, p_j) . If we query the value of this form on the edge (j, i) we should get $-f_{ij}$.

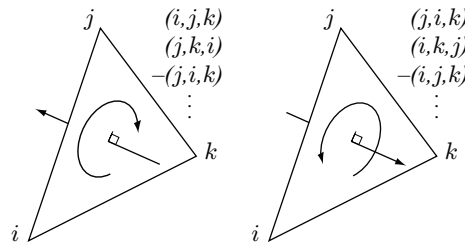


Figure 2: All permutations of a triple (i, j, k) refer to the same triangle, and the sign of the permutation determines the orientation.

Hence every tuple must be given a sign indicating whether it agrees (+) or disagrees (−) with the intrinsic orientation of the simplex. Given a set of integers representing a simplex, there are two equivalence classes of orderings of the given tuple: the even and odd permutations of the integers in question. These two equivalence classes correspond to the two possible orientations of the simplex (see Fig. 2).

Note that assigning a sign to any one alias (*i.e.*, the representative) implicitly assigns a sign to all other aliases. Let us assume for a moment that the sign of all representatives is known. Then the sign S of an arbitrary tuple t , with representative r , is

$$\mathbf{S}(t) = \begin{cases} \mathbf{S}(r) & \text{if } t \text{ is in the same equivalence class as } r \\ -\mathbf{S}(r) & \text{if } t \text{ is in the opposite equivalence class.} \end{cases}$$

More formally, let P be the permutation that permutes t into r (*i.e.*, $r = P(t)$). Then

$$\mathbf{S}(t) = \mathbf{S}(P)\mathbf{S}(P(t)).$$

(Here $\mathbf{S}(P)$ denotes the sign of the permutation P with +1 for even and −1 for odd permutations.)

All that remains, then, is to choose an intrinsic orientation for each simplex and set the sign of the representative alias accordingly. In general the assignment of orientations is arbitrary, as long as it is consistent. For all subsimplices we choose the representative to be positively oriented, so that the right-hand-side of the above expression reduces to $\mathbf{S}(P)$. For top-level simplices (tets in 3D, triangles in 2D), we use the convention that a positive volume corresponds to a positively oriented simplex. We therefore require a volume form which, together with an assignment of points to vertices, will allow us to orient all tets. Recall that a volume form accepts three (for 3D; two for 2D) vectors and returns either a positive or negative number (assuming the vectors are linearly independent). So the sign of a 4-tuple is:

$$\mathbf{S}(i_0, i_1, i_2, i_3) = \mathbf{S}(\text{Vol}(p_{i_1} - p_{i_0}, p_{i_2} - p_{i_0}, p_{i_3} - p_{i_0})).$$

4 The Boundary Operator

The *faces* of a k -simplex are the $(k - 1)$ -simplices that are incident on it, *i.e.*, the subset of one lower dimension. Every k -simplex has $k + 1$ faces. Each face corresponds to removing one integer from the tuple, and the relative orientation of the face is $(-1)^i$ where i is the index of the integer that was removed. To clarify:

- The faces of a tet $+(t_0, t_1, t_2, t_3)$ are $-(t_0, t_1, t_2)$, $+(t_0, t_1, t_3)$, $-(t_0, t_2, t_3)$, and $+(t_1, t_2, t_3)$.
- The faces of a triangle $+(f_0, f_1, f_2)$ are $+(f_0, f_1)$, $-(f_0, f_2)$, and $+(f_1, f_2)$.
- The faces of an edge $+(e_0, e_1)$ are $-(e_0)$ and $+(e_1)$.

We can now define the boundary operator ∂ which maps simplices to their faces. Given the set of tets \mathbf{T} we define $\partial^3 : \mathbf{T} \rightarrow \mathbf{F}^4$ as

$$\partial^3(+ (i_0, i_1, i_2, i_3)) = \{-(i_0, i_1, i_2), +(i_0, i_1, i_3), -(i_0, i_2, i_3), +(i_1, i_2, i_3)\}.$$

Similarly for $\partial^2 : \mathbf{F} \rightarrow \mathbf{E}^3$ (which maps each triangle to its three edges) and $\partial^1 : \mathbf{E} \rightarrow \mathbf{V}^2$ (which maps each edge to its two vertices).

We represent these operators as sparse adjacency matrices (or, equivalently, signed adjacency lists), containing elements of type +1 and −1 only. So ∂^3 is implemented as a matrix of size $|\mathbf{F}| \times |\mathbf{T}|$ with 4 non-zero elements per column, ∂^2 an $|\mathbf{E}| \times |\mathbf{F}|$ matrix with 3 non-zero elements per column, and ∂^1 a $|\mathbf{V}| \times |\mathbf{E}|$ matrix with 2

non-zero elements per column (one +1 and one −1). The transposes of these matrices are known as the *coboundary* operators, and they map simplices to their *cofaces*—neighbor simplices of one higher dimension. For example, $(\partial^2)^T$ maps an edge to the “pin-wheel” of triangles incident on that edge.

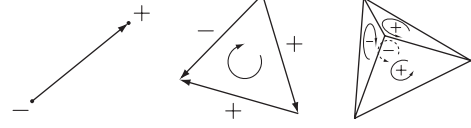


Figure 3: The boundary operator identifies the faces of a simplex as well as their relative orientations. In this illustration, arrows indicate intrinsic orientations and signs indicate the relative orientation of a face to a parent.

These matrices allow us to iterate over the faces or cofaces of any simplex, by walking down the columns or across the rows, respectively. In order to traverse neighbors that are more than one dimension removed (*i.e.*, the tets adjacent to an edge or the faces adjacent to a vertex) we simply concatenate the appropriate matrices, but without the signs. (If we kept the signs in the matrix multiplication any such consecutive product would simply return the zero matrix reflecting the fact that the boundary of a boundary is always empty.)

5 Construction

Although we still need a few auxiliary wrapper and iterator data structures to provide an interface to the mesh elements, the simplex lists and boundary matrices contain the entirety of the topological data of the mesh. All that remains, then, is to fill in this data.

We read in our mesh as a list of (x, y, z) vertex positions and a list of 4-tuples specifying the tets. Reading the mesh in this format eliminates the possibility of many non-manifold scenarios; for example, there cannot be an isolated edge that does not belong to a tet. We assume that all integers in the range $[0, n)$ appear at least once in the tet list (this eliminates isolated vertices), and no integer outside of this range is present.

Once \mathbf{T} is read in, building \mathbf{E} and \mathbf{F} is trivial; for each tuple in \mathbf{T} , append all subsets of size 2 and 3 to \mathbf{E} and \mathbf{F} respectively. We must be sure to avoid duplicates, either by using a unique associative container, or by sorting the list afterward and removing duplicates. Then the boundary operator matrices are constructed as follows:

```
for each simplex s
  construct a tuple for each face f of s
  as described in Section 4
  determine the index i of f by locating
  its representative
  set the entry of the appropriate matrix
  at row i, column s to S(f)
```

Figure 4 shows a complete example of a mesh and its associated data structure.

6 DEC Operators

Now we discuss the implementation of the two most commonly used DEC operators: the exterior derivative and the Hodge star. As we will see, in the end these also amount to nothing more than sparse matrices that can be applied to our form vectors.

6.2.1 Calculating Dual Volumes

So far the entire implementation has been in terms of the combinatorics of the mesh, but when constructing the Hodge star we must finally introduce the geometry. After all, the purpose of the Hodge star is to capture the metric. The volumes of the primal simplices are straightforward: 1 for vertices, length for edges, area for triangles, and volume for tetrahedra. The dual volumes are similarly defined, but in order to avoid constructing the graph of the dual mesh explicitly, we calculate the dual volumes as follows.

If we use the circumcentric realization of the dual mesh (*i.e.*, dual vertices are at the circumcenters of the associated tets), we can exploit the following facts when calculating the dual volumes.¹

- A dual edge (dual of a primal triangle t) is linear, is normal to t , and is collinear with the circumcenter of t (though the line segment need not necessarily pass through t).
- A dual polygon (dual of a primal edge e) is planar, is orthogonal to e , and is coplanar with the center of e (though it need not intersect e).
- A dual cell (dual of a primal vertex v) is the convex intersection of the half-spaces defined by the perpendicular bisectors of the edges incident on v .

Just as with primal vertices, the volume of a dual vertex is defined to be 1. For the others, we can conceptually decompose each cell into pieces bounded by lower dimensional cells, and sum the volumes of the pieces. For example, a dual polyhedron can be seen as the union of some number of pyramids, where the base of each pyramid is a dual polygon and the apex is the primal vertex. Similarly, a dual polygon can be seen as a union of triangles with dual edges at the bases, and dual edges can be seen as a union of (two) line segments with dual vertices at the bases. The following pseudocode illustrates how the volumes are calculated.

```

vec3 C( Simplex s ); // gives the circumcenter of s

// Initialize all dual volumes to 0.

// Dual edges
for each primal triangle f
  for each primal tet  $t_f$  incident on f
     $b \leftarrow t_f.\text{dualVolume} // 1$ 
     $h \leftarrow \|C(f) - C(t_f)\|$ 
     $f.\text{dualVolume} \leftarrow f.\text{dualVolume} + \frac{1}{3}bh$ 

// Dual polygons
for each primal edge e
  for each primal triangle  $f_e$  incident on e
     $b \leftarrow f_e.\text{dualVolume}$ 
     $h \leftarrow \|C(e) - C(f_e)\|$ 
     $e.\text{dualVolume} \leftarrow e.\text{dualVolume} + \frac{1}{2}bh$ 

// Dual polyhedra
for each primal vertex v
  for each primal edge  $e_v$  incident on v
     $b \leftarrow e_v.\text{dualVolume}$ 
     $h \leftarrow \|C(v) - C(e_v)\|$ 
     $v.\text{dualVolume} \leftarrow v.\text{dualVolume} + \frac{1}{3}bh$ 

```

Note that, even when dealing with the geometry of the mesh, this part of the implementation still generalizes trivially to arbitrary dimension.

¹ Circumcentric duals may only be used if the mesh satisfies the Delaunay criterion. If it does not, a barycentric dual mesh may be used. However, care must be taken if a barycentric dual mesh is used, as dual edges are no longer straight lines (they are piecewise linear), dual faces are no longer planar, and dual cells are no longer necessarily convex.

7 Summary

All the machinery discussed above can be summarized as follows:

- k -forms as well as the Hodge star are represented as vectors of length $|\mathbf{V}|$, $|\mathbf{E}|$, $|\mathbf{F}|$, and $|\mathbf{T}|$;
- the discrete exterior derivative is represented as (transposes of) sparse adjacency matrices containing only entries of the form $+1$ and -1 (and many zeros); the adjacency matrices are of dimension $|\mathbf{V}| \times |\mathbf{E}|$ (boundary of edges), $|\mathbf{E}| \times |\mathbf{F}|$ (boundary of triangles), and $|\mathbf{F}| \times |\mathbf{T}|$ (boundary of tets).

In computations these matrices then play the role of operators such as grad, curl, and div and can be composed to construct operators such as the Laplacian (and many others).

While the initial setup of these matrices is best accomplished with associative containers, their final form can be realized with standard sparse matrix representations. Examples include a compressed row storage format, a vector of linked lists (one linked list for each row), or a two dimensional linked list (in effect, storing the matrix and its transpose simultaneously) allowing fast traversal of either rows or columns. The associative containers store integer tuples together with orientation signs. For these we suggest the use of sorted integer tuples (the canonical representatives of each simplex). Appropriate comparison operators needed by the container data structures simply perform lexicographic comparisons.

And that's all there is to it!

Acknowledgments This work was supported in part through a James Irvine Fellowship to the first author, NSF (DMS-0220905, DMS-0138458, ACI-0219979), DOE (W-7405-ENG-48/B341492), nVidia, the Center for Integrated Multiscale Modeling and Simulation, Alias, and Pixar.

References

JOY, K. I., LEGAKIS, J., AND MACCRACKEN, R. 2002. *Data Structures for Multiresolution Representation of Unstructured Meshes*. Springer-Verlag, Heidelberg, Germany.