

# Computational Meshing

Practical Application of PETSc's DMplex

**Matthew G. Knepley**



**University at Buffalo**

Department of Computer Science and Engineering

University At Buffalo

November 4, 2022

I dedicate these notes to my wonderful wife Margarete, without whose  
patience and help they could never have been written.

## Acknowledgements

TBD



# Contents

<b>I</b>	<b>Representations</b>	<b>9</b>
<b>1</b>	<b>Topology</b>	<b>11</b>
1.1	Conforming Topology . . . . .	11
1.1.1	The Hasse Diagram . . . . .	13
1.1.2	Mesh Interpolation . . . . .	16
1.1.3	Orientation . . . . .	16
1.1.4	Periodicity . . . . .	37
1.1.5	CAD Interface . . . . .	37
1.2	Nonconforming Topology . . . . .	37
1.2.1	The Parent Tree . . . . .	37
1.2.2	Anchors and Constraints . . . . .	37
1.3	Submeshes . . . . .	37
1.4	Parallelism . . . . .	37
1.4.1	Distribution . . . . .	37
<b>2</b>	<b>Functions</b>	<b>39</b>
2.1	PetscSection . . . . .	39
2.1.1	Constraints . . . . .	39
2.1.2	Symmetries . . . . .	39
2.2	Parallelism . . . . .	40
2.2.1	Local and Global Sections . . . . .	40
2.2.2	Data Distribution . . . . .	40
2.3	Periodicity . . . . .	40
2.4	Global Basis Transformation . . . . .	40
2.5	Geometry . . . . .	40
2.6	Projection . . . . .	40
2.6.1	Interpolation . . . . .	40
2.6.2	$L_2$ projection . . . . .	41
<b>II</b>	<b>Transformations</b>	<b>43</b>
<b>3</b>	<b>General Transformations</b>	<b>45</b>
3.1	Definition . . . . .	47

3.2	Group Action . . . . .	50
3.3	Numbering . . . . .	51
3.4	Implementation . . . . .	52
<b>4</b>	<b>Interpolating</b>	<b>55</b>
4.1	Serial Algorithm . . . . .	55
4.2	Parallel algorithm . . . . .	57
<b>5</b>	<b>Extracting</b>	<b>61</b>
5.1	Filtering . . . . .	61
5.2	Submeshes . . . . .	61
<b>6</b>	<b>Extruding</b>	<b>63</b>
6.1	Simple Extrusion . . . . .	63
6.1.1	Coordinates . . . . .	66
6.2	Embedded Extrusion . . . . .	67
6.2.1	Labeling . . . . .	68
6.2.2	Construction . . . . .	68
<b>7</b>	<b>Refining</b>	<b>71</b>
7.1	Regular refinement . . . . .	71
7.1.1	Converting cell types . . . . .	74
7.1.2	Boundary Layers . . . . .	76
7.1.3	Snapping to structure . . . . .	76
7.2	Adaptive refinement . . . . .	76
7.2.1	Plaza . . . . .	76
7.2.2	p4est . . . . .	77
7.2.3	ParMMG . . . . .	77
<b>8</b>	<b>Coordinate Transformations</b>	<b>79</b>
8.1	Coordinate Representation . . . . .	79
8.2	Direct Modification . . . . .	79
8.3	Projection . . . . .	80
<b>III</b>	<b>Applications</b>	<b>83</b>
<b>9</b>	<b>Crustal Dynamics</b>	<b>85</b>
<b>10</b>	<b>Low Mach Flow</b>	<b>87</b>
<b>Appendix A</b>	<b>Creating DMPlex Meshes</b>	<b>91</b>
A.1	Working with files . . . . .	91
A.2	Working with Shapes . . . . .	93
A.3	Working in Parallel . . . . .	98
<b>Appendices</b>		

*CONTENTS*

7

**Index**

**101**





Part I

Representations



# Chapter 1

# Topology

*Point set topology is a disease from which the human race will soon recover.*

— Henri Poincare

## 1.1 Conforming Topology

In the last century, at the birth of topology, two main organizing concepts emerged for the new discipline, the *complex* and the *topological space*, both of which are attempts to describe mathematically the intuitive idea of a geometrical figure. Since Euclid, a figure has been thought of as a heterogeneous collection of elements (points, lines, planes, ...), or configuration, arranged according to attachment rules. These will become our complexes, following the line of thinking from synthetic geometry. The other point of view is that a figure is an infinite collection of homogeneous elements, a point set, organized to form a geometrical figure. This is usually done by introducing a coordinate system, metric, and idea of neighborhoods. The synthesis of these two lines of thought comes about through the work of Brouwer <sup>1</sup>.

We will take the combinatorial approach to topology, meaning that we will cut space into a finite number of pieces, rather than imaging it as being composed an infinite sets of points. This is a much more natural framework for understanding and manipulating descriptions of space on a computer. We will call the fundamental units of this division a *k-cell*, where *k* indicates the dimension of a cell. We cannot change the dimension of a cell by a continuous mapping, so this classification makes sense here. In fact, all *k*-cells are infinitesimally close to a *k* dimensional polyhedron ([Dimension theory 2021](#)), so that we can always think of breaking up our domain into *k*-cells which have well-defined faces. We will call a decomposition *conforming* if any two *k*-cells are disjoint,

---

<sup>1</sup>A beautiful description of these developments appears in the early book of Paul Alexandroff [Alexandroff 1932](#)

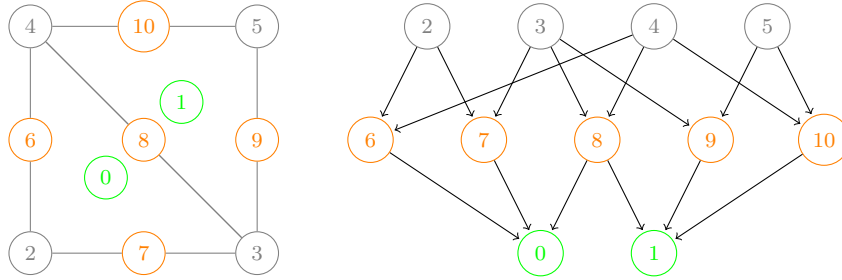


Figure 1.1: Two adjacent triangles, and the corresponding Hasse diagram.

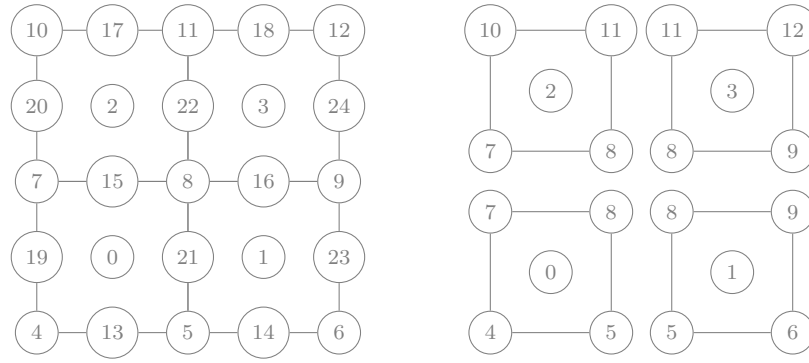


Figure 1.2: A square decomposed into cell closure patches.

or their intersection defines a face which is a  $(k - 1)$ -cell. An example is shown in Fig. 1.1, where two 2-cells (triangles) intersect in a 1-cell (edge).

Discussion of cone and support with pictures

We commonly think of building up a mesh from pieces which contain their closure, in a similar way to a jigsaw puzzle. For example, in Fig. 1.2, we construct a square mesh from four smaller squares. The boundaries of these smaller pieces overlap when we put them together, and we eliminate the redundant boundary points. However, a complementary, or *dual*, point of view would see the square mesh as built up from a central vertex attached to four edges and four square interiors, four edge vertices attached to two edges and two square interiors, and four corner vertices attached to two edges and one square interior. When we put those pieces together, the edges and interiors overlap, and we discard the redundant points, just as we did in the case for boundaries.

This dual point of view becomes more important when we are forced to divide our mesh, rather than keep a single, coherent representation. For example, if we run in parallel, we would need to store part of the mesh on each machine, rather than replicating the entire mesh, in order to be memory scalable. Suppose that we choose to divide our square into the four smaller squares, and store each one on a different process. On each process we would have the cone of

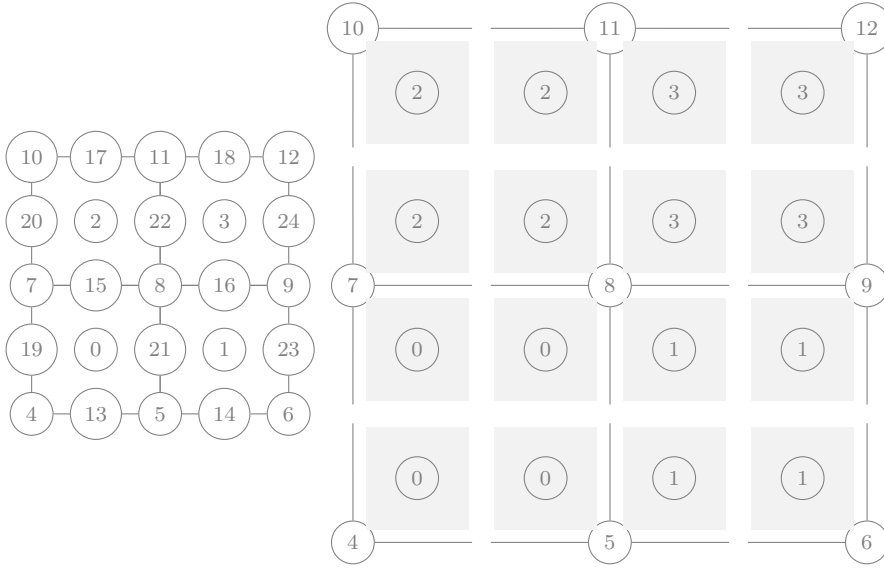


Figure 1.3: A square decomposed into vertex star patches.

each stored point, but the supports would be incomplete. For example, the support of each interior edge contains two squares, but only one square would be available on that process. The solution might seem simple. Why not just add the off-process square to the support of that edge? However, if we do that, then we have a point (the new square) with an incomplete cone. We could add the boundary of the new square, but that would mean adding points with incomplete supports. It now becomes clear, that pieces we extract from a mesh can have either complete cones or complete supports, *but not both*. In Plex, we guarantee the completeness of local cones, but not supports, because people naturally think of pieces containing their boundary. One could equally well construct Plex with complete supports, using vertex star pieces, but this has not yet been implemented.

### 1.1.1 The Hasse Diagram

Let  $P_k(p, q)$  be the relation “there exists a directed path of length  $k$  from  $p$  to  $q$  in the Hasse diagram”, and let  $P$  be the union of all these relations

$$P = \bigcup_k P_k. \tag{1.1}$$

We can define the cone and support operations as a duality relation

$$q \in \text{cone}(p) \iff P_1(p, q) \iff p \in \text{supp}(q). \tag{1.2}$$

And we can generalize this to transitive closures,

$$q \in \mathfrak{cl}(p) \iff P(p, q) \iff p \in \mathfrak{st}(q). \quad (1.3)$$

Once we have the Hasse diagram, we can identify the boundary of any  $k$ -cell  $p$ . It is the set of mesh points given by  $\mathfrak{cl}(p) - p$ . However, if we decompose a manifold into  $k$ -cells, we do not have enough information to orient it. For example, if I triangulated a surface, I would not be able to calculate a unique normal because I would not know the order of vertices around each triangle. Seen from a combinatorial points of view, the formal sum of a boundary should vanish, but I do not know what weights to put on the vertices in the closure.

What we need is some way to distinguish one configuration of a  $k$ -cell from another, and by *configuration* we will mean the order of points in its cone. Taken transitively, this will impose an order on the closure of any point. However, this order might not be what we want. For example, we would like the boundary sum to vanish when applied to something that is already a boundary, which is what we usually mean by orienting a manifold. This will only happen if we are allow to orient mesh point differently, depending on what cell they are attached to. A simple example will help illustrate the abstract point.

The cone of triangle 0 is the edges  $\{6, 7, 8\}$ , and we can imagine that these edges have cones  $\{4, 2\}$ ,  $\{2, 3\}$ , and  $\{3, 4\}$ . Thus, our formal boundary sum is given by

$$\text{bd}(0) = 6 + 7 + 8$$

and is just another form of the cone. The boundary of a boundary should vanish, and it does

$$\begin{aligned} \text{bd}(\text{bd}(0)) &= \text{bd}(6) + \text{bd}(7) + \text{bd}(8) \\ &= (4 - 2) + (2 - 3) + (3 - 4) \\ &= \emptyset \end{aligned}$$

if we orient the edges as suggested. However, let us try the same thing with triangle 1, whose cone is  $\{9, 10, 8\}$ , and the edges have cones  $\{3, 5\}$ ,  $\{5, 4\}$ , and  $\{3, 4\}$ . Then the boundary of 1 is

$$\text{bd}(1) = 9 + 10 + 8$$

and the second boundary operation gives

$$\begin{aligned} \text{bd}(\text{bd}(1)) &= \text{bd}(9) + \text{bd}(10) + \text{bd}(8) \\ &= (3 - 5) + (5 - 4) + (3 - 4) \\ &= (3 - 4) + (3 - 4) \\ &\neq \emptyset. \end{aligned}$$

In this simple example, the problem is clear. Triangle 0 would like edge 8 to be directed north-east, but triangle 1 would like it to be oriented south-west. The

solution is to allow triangle 1 to attach edge 8 with the opposite orientation. Then we would have,

$$\begin{aligned}
 \text{bd}(\text{bd}(1)) &= \text{bd}(9 + 10 - 8) \\
 &= \text{bd}(9) + \text{bd}(10) - \text{bd}(8) \\
 &= (3 - 5) + (5 - 4) - (3 - 4) \\
 &= (3 - 4) + (4 - 3) \\
 &= \emptyset.
 \end{aligned}$$

This use of just  $+/-$  is fine for edges, but if we have faces with symmetry groups that are more complex than  $S_2$ , we must allow a more sophisticated system for labeling the possible ways of attaching a  $(k - 1)$ -cell face to a  $k$ -cell. You can think of these labels as labeling the edges in the Hasse diagram.

Suppose we take a  $k$ -cell in the canonical configuration. We could generate an alternative cell by changing the arrangement of the faces, and label each arrangement with a number so that we can tell them apart. We will call this number the *orientation* for the cell. It will turn out to be more fruitful to think of labeling, not the configuration itself, but the transformation that gets us from the canonical configuration to the alternative one we are labeling. This gets us to the idea of *group transformations*, where now we label all the members of the group by our orientation number. Thinking this way allows us to compose one transformation with another, so that we have a prescription for transforming our cells. In the next section, we will walk through examples to show how this works in practice.

### Continuous Maps

There are many equivalent definitions of continuity. The simplest one from grade school is that the graph of a continuous function can be drawn without lifting your pencil. A more sophisticated version of this, referred to as an  $\epsilon - \delta$  definition, is that given an target point of the map, we can find another point as close to it as we please (distance  $\epsilon$ ) by taking another source point sufficiently close (distance  $\delta$ ) to our original source point. We can imagine this as a little ball of radius  $\delta$  around each source point which maps to another ball of radius  $\epsilon$  around a target point. This suggest another definition, that continuous functions map open sets to other open sets. Embedded in all these definitions is the idea of being "close to" something else, which is at the heart of topology. If two points are close to each other in the source space, their images should be close to each other in the target space in order to preserve the topology.

If we think of this in terms of simplicial decompositions. continuous maps should preserve cells, since all points in a cell are considered close, and in fact algebraic topology sees the cell as a single "point". Moreover, the boundary of a cell, or those cells covering it, should also cover it in the target space. This means that the Hasse diagram is invariant under continuous maps.

Continuous maps are important because they preserve topological characteristics, famously embodied in *invariance theorems*, meaning that if a property

holds for one simplicial decomposition, it will hold for all such decompositions. The most famous such theorem in Brouwer's demonstration of the invariance of dimension: if an  $n$ -dimensional complex appears as the decomposition of a polyhedron  $P$ , then every decomposition of  $P$  is  $n$ -dimensional, as well as every  $P'$  that is homeomorphic to  $P$  meaning it was produced by a continuous map.

### 1.1.2 Mesh Interpolation

By *mesh interpolation*, we mean the process of automatically creating  $k$ -faces in a mesh that formerly consisted solely of cells and vertices. For example, in the 3D dimensional mesh, this would mean creating both edges and faces (of whatever type is required). Faces of any type can be created directly in a Plex, but we can only interpolate faces that have first been defined for certain cell types. The list of all cell types which support interpolation is shown in Table 1.1, along with the cell abbreviations and canonical figure. Notice that in the figures, we number the vertices and edges with their point number in the Plex, rather than the ordinal number.

Now, we must define the faces that we are able to interpolate. We will define a face as the ordered set of vertices in its closure. Note that this imposes a strong constraint on interpolation. The closure after interpolation must preserve the order of vertices. In Table 1.2, we give the ordered set of vertices for each face such that the face normal is outward. We number the vertices using ordinal numbers, where the vertex numbering is just shifted to zero from the Plex ordering in Table 1.1. This is to mirror the code, in `DMPlexGetRawFaces_Internal()`, which uses these numbers as offsets, so that you can index into the cone of a cell from an uninterpolated mesh.

In Chapter 4, we will present algorithms for interpolating a mesh in parallel. This operation has complexity linear in both the input and output meshes, noting the number of faces and edges is linear in the number of cells and vertices by Euler's formula. This will be convenient when reading in meshes from other sources, or when using a mesh generator library.

### 1.1.3 Orientation

We can naturally conceive of a geometric figure being built up out of smaller pieces, and this notion is at the heart of computational meshing. We divide a complicated manifold into pieces which are simpler to understand and manipulate. We will think of each simple piece as containing its boundary, and thus we are led to think about closures as being the basic pieces of a mesh. Each piece will be a complex, in the same sense as we defined it above, as will the combination of all the pieces. In order to understand the process of putting the pieces together, we would like to understand the symmetry properties of the pieces themselves. We will call each possible different arrangement of a complex an *orientation*, and assign each one a number. We can also think of these numbers as referring to a specific element of the symmetry group for this complex.



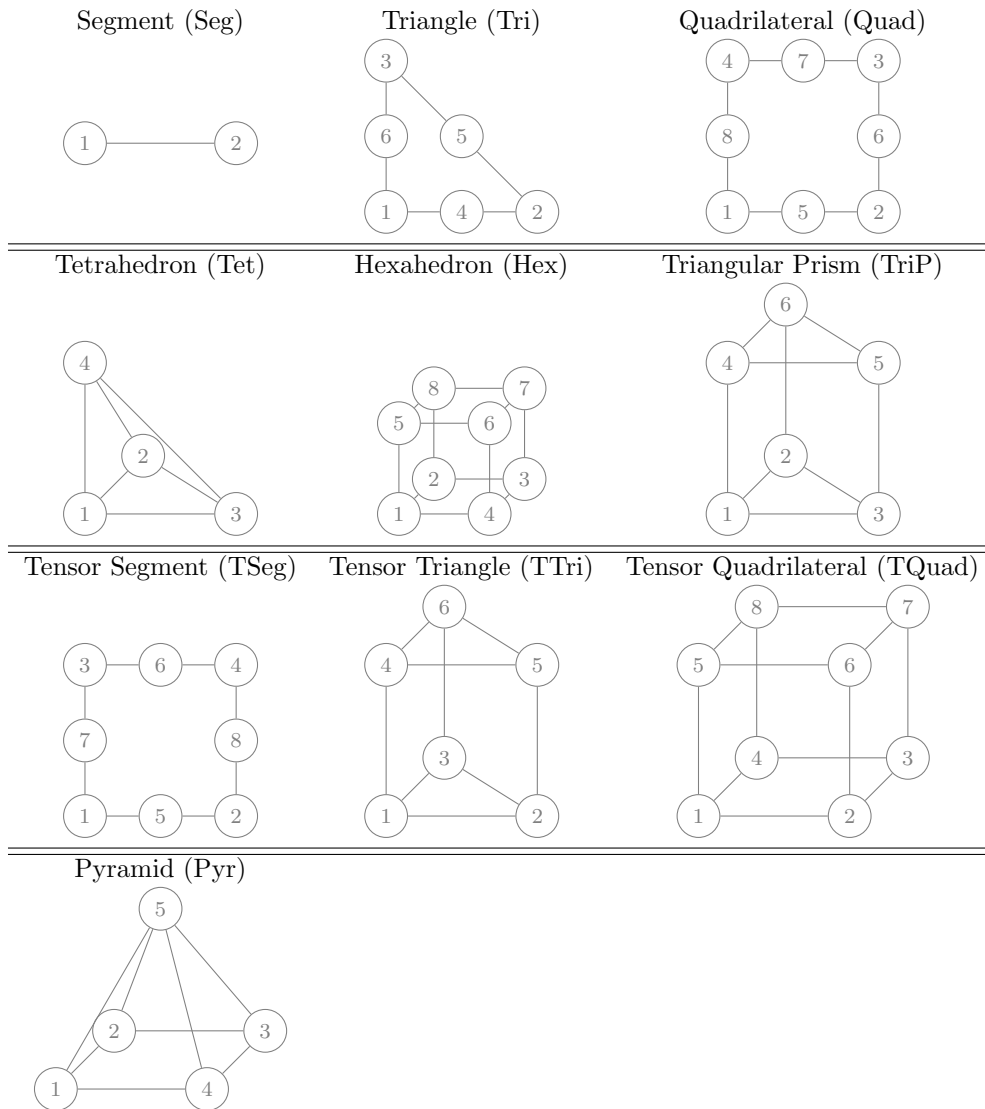
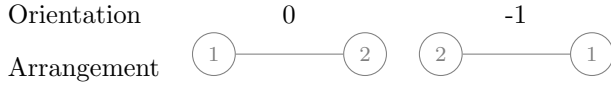


Table 1.1: The cell types which support interpolation.

Cell Type	Face 0	Face 1	Face 2	Face 3	Face 4	Face 5
Seg	{0}	{1}				
	Pnt	Pnt				
TPnt	{0}	{1}				
	Pnt	Pnt				
Tri	{0, 1}	{1, 2}	{2, 0}			
	Seg	Seg	Seg			
Quad	{0, 1}	{1, 2}	{2, 3}	{3, 0}		
	Seg	Seg	Seg	Seg		
TSeg	{0, 1}	{2, 3}	{0, 2}	{1, 3}		
	Seg	Seg	TPnt	TPnt		
Tet	{0, 1, 2}	{0, 3, 1}	{0, 2, 3}	{2, 1, 3}		
	Tri	Tri	Tri	Tri		
Hex	{0, 1, 2, 3}	{4, 5, 6, 7}	{0, 3, 5, 4}	{2, 1, 7, 6}	{3, 2, 6, 5}	{0, 4, 7, 1}
	Quad	Quad	Quad	Quad	Quad	Quad
TriP	{0, 1, 2}	{3, 4, 5}	{0, 2, 4, 3}	{2, 1, 5, 4}	{1, 0, 3, 5}	
	Tri	Tri	Quad	Quad	Quad	
TTri	{0, 1, 2}	{3, 4, 5}	{0, 1, 3, 4}	{1, 2, 4, 5}	{2, 0, 5, 3}	
	Tri	Tri	TSeg	TSeg	TSeg	
TQuad	{0, 1, 2, 3}	{4, 5, 6, 7}	{0, 1, 4, 5}	{1, 2, 5, 6}	{2, 3, 6, 7}	{3, 0, 7, 4}
	Quad	Quad	TSeg	TSeg	TSeg	TSeg
Pyr	{0, 1, 2, 3}	{0, 3, 4}	{3, 2, 4}	{2, 1, 4}	{1, 0, 4}	
	Quad	Tri	Tri	Tri	Tri	

Table 1.2: Definition of faces for each cell type, using ordinal numbering

Table 1.3: The dihedral group  $D_2$  for the segment

The simplest mesh point is a vertex, which has a single transformation, the identity, since it has no faces. We will always label the identity transformation by 0. In the code, the cell type of a vertex is `DM_POLYTOPE_POINT`, but in this text we will indicate this using `POINT`, and similarly for all the members of that enumeration type. If we combine two vertices together, we get an edge, `SEGMENT`. The segment has two possible arrangements, the canonical one (0) and one with the vertices flipped (-1), shown in Table 1.3. The group of transformations of a regular  $n$ -gon is called the *dihedral group* on  $n$  elements,  $D_n$ , so that the group for the segment is  $D_2$ . This group naturally separates into rotations and reflections, and for  $D_2$  we have only the reflection. In our labeling, we will indicate the transformation you obtain by reflecting an operation  $o$  as  $-(o+1)$ . Thus, when we reflect the identity transformation 0, we get  $-(0+1) = -1$ . Of course other numberings are possible, and can be easily achieved once our code is driven by the group multiplication table.

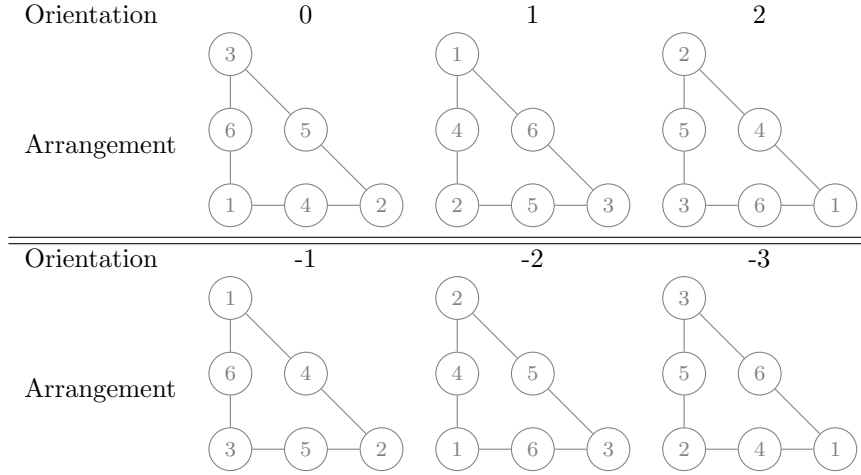
Notice that we could also orient the star in the same way that we have oriented the closure. The shapes are not as familiar as the boundaries of simple shapes shown below, but they are not esoteric. In two dimensions, the edges around a vertex transform as the dihedral group  $D_n$ , and similarly for the faces around an edge in 3D, whereas faces will always have  $S_2$  as a symmetry group. The edges around a vertex in 3D will represent one of the [finite spherical symmetry groups](#). These should not be surprising, since we have a duality between these groups and the cone symmetry groups. However, we will not pursue this right now, since our principal numerical methods are formulated on the closure of cells.

Extensibility of cell types in Plex: Go over everything needed to register another cell type. Should rewrite code to allow user-defined cell types.

Below, we will show the possible arrangements, and the associated orientation, for all  $k$ -cells that Plex can interpolate. These pictures were generated and checked using Plex tutorial [ex11](#) in the PETSc source tree. The `asciilinux` output format for Plex produces TikZ pictures that can be embedded in a larger document, as we have done here.

### Triangle

Our next example, the group  $D_3$  for the symmetries of the equilateral triangle, will allow us to look at the group table. In Table 1.4, the canonical arrangement starts with edge 4, and proceeds counter-clockwise with edges 5 and 6. An additional complication here is that the faces are also oriented. The canonical orientation of these edges is (1, 2), (2, 3), and (3, 1), so that they are also counter-clockwise. Orientation 1 produces the arrangement shown in the table,

Table 1.4: The dihedral group  $D_3$  for the triangle

namely that we begin on edge 5 with the same counter-clockwise ordering. Similarly, for Orientation 2, we start on edge 6. We can also interpret the orientation as talking about the progression of vertices, as it follows from the edges.

Now we consider the reflected orientations. For the identity, it is  $-(0+1) = -1$ . This corresponds to reversing the order of the edges *and* reversing the order of each edge in the sequence. Therefore when we reflect a cell orientation, we must also reflect the orientation of the faces. However, instead of edge 6, we begin at edge 5, which reversed has vertices (3, 2), and proceed to edge 4, now (2, 1), and finally edge 6 as (1, 3). This produces the vertex sequence (3, 2, 1), which is what we expect from reflecting (1, 2, 3). Thus our convention is designed to produce certain vertex orderings in the closure, rather than face orderings. We can interpret the orientation as a permutation of the vertices. Let us take  $e$  as the identity permutation,  $a$  as swapping the first two vertices, and  $b$  as swapping the last two vertices. Then we can write the group operation table for our orientations as follows

Orientation	Permutation	Operation
-3	0, 2, 1	$b$
-2	2, 1, 0	$aba$
-1	1, 0, 2	$a$
0	0, 1, 2	$e$
1	1, 2, 0	$ba$
2	2, 0, 1	$ab$

where the numbers refer to faces in the cone, not vertices in the closure. We see that the reflections have an odd number of swaps, whereas the rotations have an even number. From this it is also clear that  $D_3$  has two generators,  $a$  and  $b$ .

Moreover, we can write the group multiplication or *Cayley Table*, which shows how orientation operations combine

	-3	-2	-1	0	1	2
-3	0	1	2	-3	-2	-1
-2	2	0	1	-2	-1	-3
-1	1	2	0	-1	-3	-2
0	-3	-2	-1	0	1	2
1	-1	-3	-2	1	2	0
2	-2	-1	-3	2	0	1

### Quadrilateral

We can look at the dihedral group for the square,  $D_4$ , for an example that does not contain all permutations of the vertices, shown in Table 1.5. This shows then same progression as the triangle, namely that the positive orientations are successive rotations, whereas the negative orientations are a rotation and a reflection.

### Tetrahedron

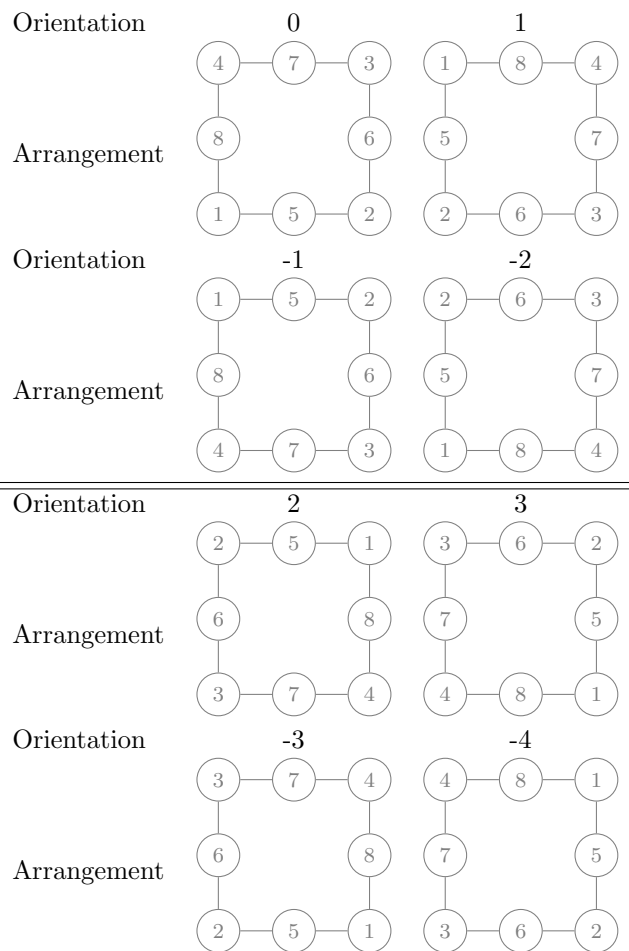
All symmetry groups of regular polytopes are finite *Coxeter groups*, and the dual polytopes have the same symmetry group. The symmetry group of the  $n$ -simplex is the symmetric group  $S_{n+1}$  of permutations of  $n+1$  elements. Thus the tetrahedron has group  $S_4$  of size  $4! = 24$ . We choose the canonical orientation to be the one with the normal to the first face pointing away from the fourth vertex, so that all faces have outward normals, as shown in Tables 1.6 and 1.7. Unfortunately, odd permutations no longer correspond to inversions of the tetrahedron, meaning that the inverse orientation cannot be one that reverses the order of all the vertices in the closure. Instead, we choose the inverse orientation to be the one that reverses the first face, as that guarantees an inversion of the tetrahedron.

### Hexahedron

The hexahedron has the symmetry group  $B_3$  which has order 48. The rotation part is isomorphic to  $S_4$  since each rotation corresponds to an arrangement of the four diagonals. Thus with reflections, it is twice as large as  $S_4$ . To reflect the hexahedron, we invert the bottom face (orientation -1), and use orientation -3 for the top face.

### Triangular Prism

The triangular prism has the symmetries of a triangle, combined with a  $180^\circ$  rotation which exchanges the top and bottom faces.

Table 1.5: The dihedral group  $D_4$  for the square

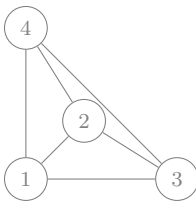
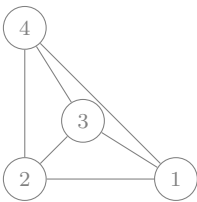
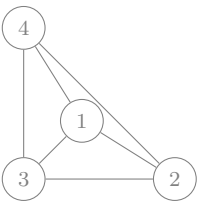
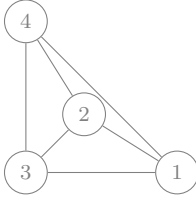
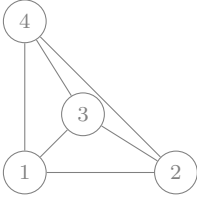
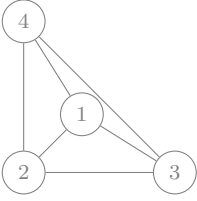
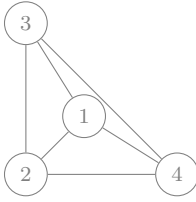
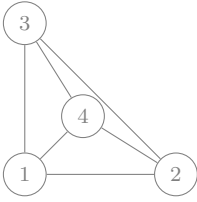
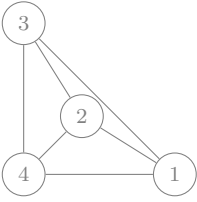
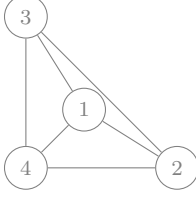
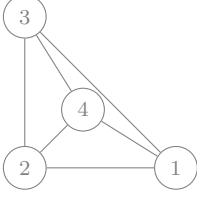
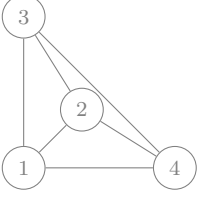
Orientation	0	1	2
Arrangement			
Orientation	-1	-2	-3
Arrangement			
Orientation	3	4	5
Arrangement			
Orientation	-4	-5	-6
Arrangement			

Table 1.6: PartI: The symmetric group  $S_4$  for the tetrahedron

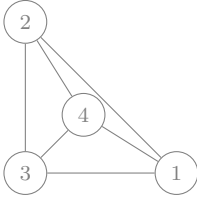
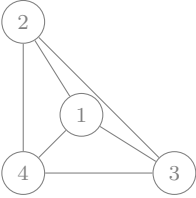
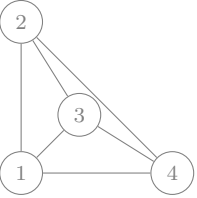
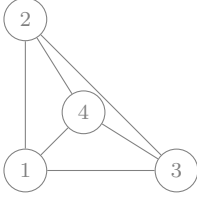
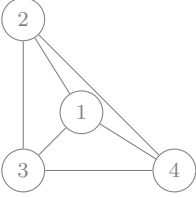
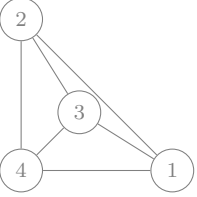
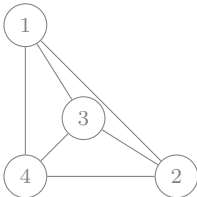
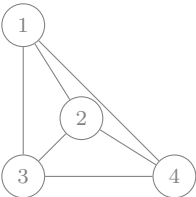
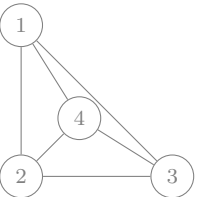
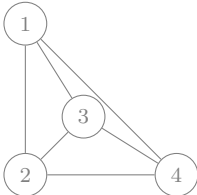
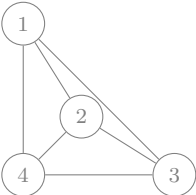
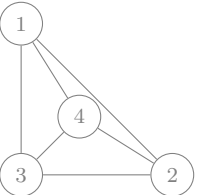
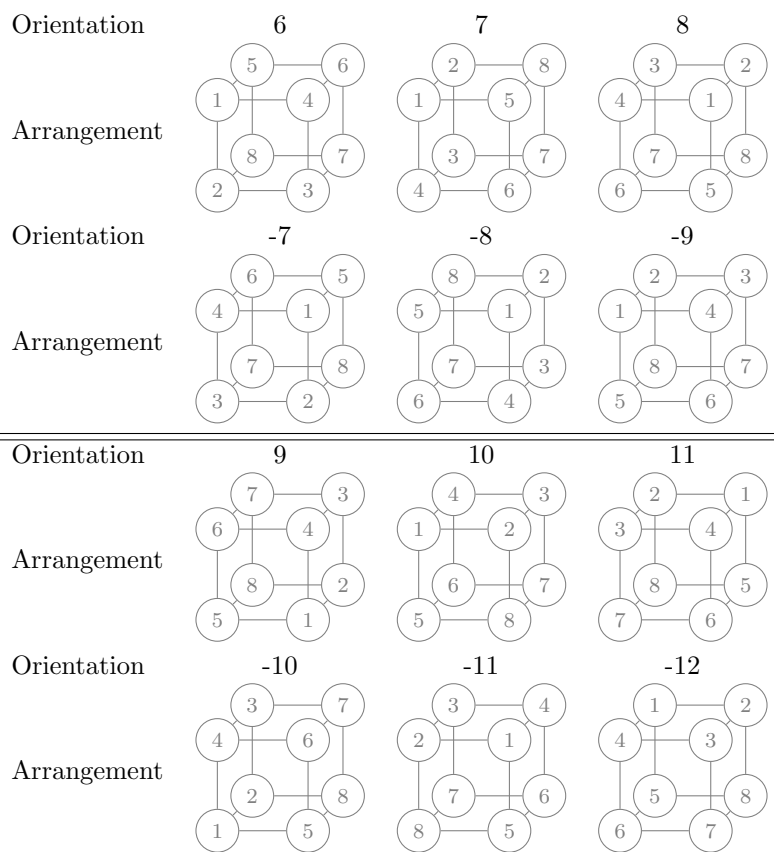
Orientation	6	7	8
Arrangement			
Orientation	-7	-8	-9
Arrangement			
Orientation	9	10	11
Arrangement			
Orientation	-10	-11	-12
Arrangement			

Table 1.7: PartII: The symmetric group  $S_4$  for the tetrahedron



Orientation	0	1	2
Arrangement			
Orientation	-1	-2	-3
Arrangement			
Orientation	3	4	5
Arrangement			
Orientation	-4	-5	-6
Arrangement			

Table 1.8: PartI: The Coxeter group  $B_3$  for the hexahedron

Table 1.9: PartII: The Coxeter group  $B_3$  for the hexahedron

Orientation	12	13	14
Arrangement			
Orientation	-13	-14	-15
Arrangement			
Orientation	15	16	17
Arrangement			
Orientation	-16	-17	-18
Arrangement			

Table 1.10: PartIII: The Coxeter group  $B_3$  for the hexahedron

Orientation	18	19	20
Arrangement			
Orientation	-19	-20	-21
Arrangement			
Orientation	21	22	23
Arrangement			
Orientation	-22	-23	-24
Arrangement			

Table 1.11: PartIV: The Coxeter group  $B_3$  for the hexahedron

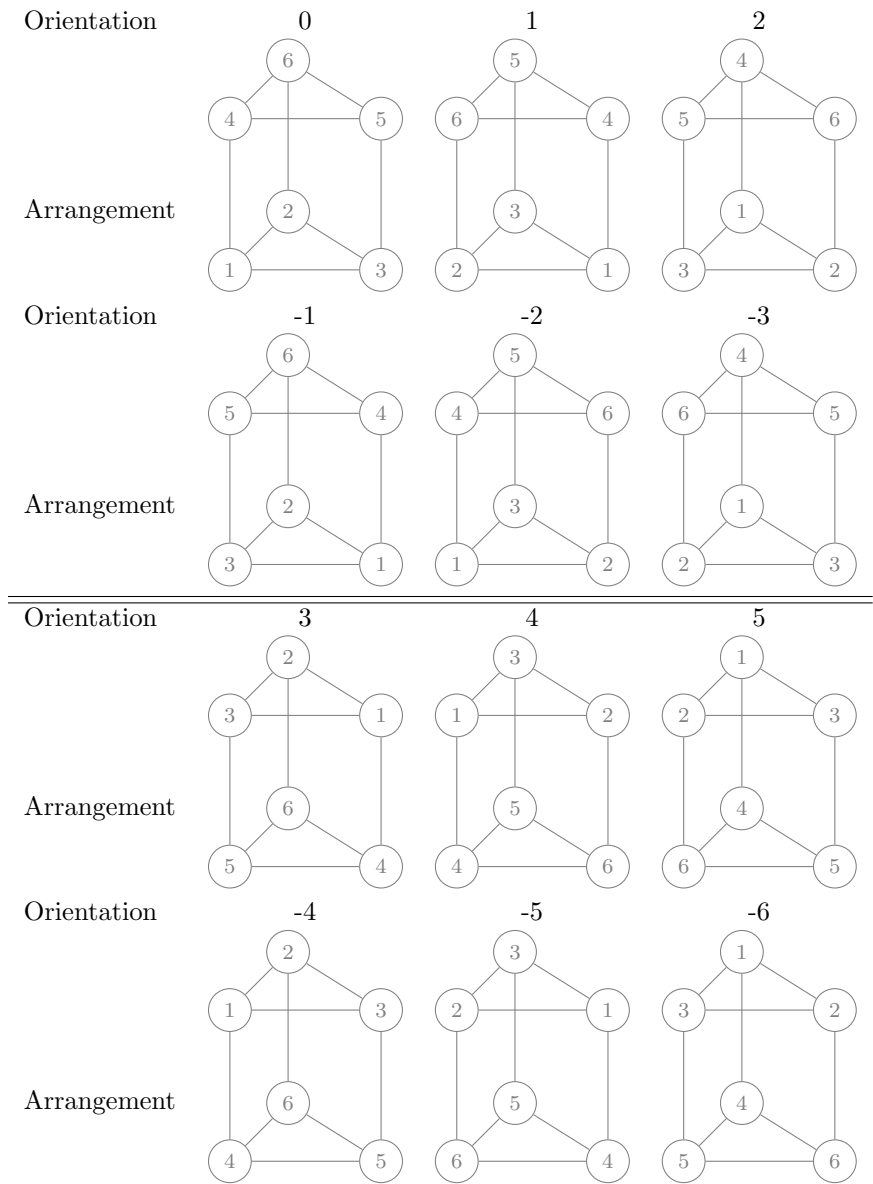


Table 1.12: The symmetry group for the triangular prism

### Pyramid

The pyramid with a square base has the same set of symmetries as its base, shown in Table 1.13.

### Tensor Segment Prism

Plex also recognizes cells that are created by taking the tensor product of two cells. The simplest of these is the tensor product of two segments. From a point-set topology point of view, this is the same as our quadrilateral. However, instead of orienting the boundary for outward normals, we use the same orientation for the two copies of the initial cell produced for the boundaries. You can see this in Table 1.14, where the top and bottom segments always have the same orientation, as do the two vertical segments (which are actually point prisms in our language). The inversion operation (r) reflects the top and bottom segments, whereas exchanging the top and bottom segments (b) does not change the volume.

### Tensor Triangular Prism

We can produce a triangular prism from the tensor product of a triangle and a segment. The symmetries are slightly different than the triangular prism above, as shown in Table 1.15, because the top and bottom faces are oriented in the same way, instead of both oriented to produce outward normals. We can describe the group as being generated by a rotation of  $120^\circ$  about the  $z$ -axis (a), an exchange of the top and bottom faces (b), and a reflection (r). We will number the orientations such that the reflection of orientation  $o$  is  $-(o + 1)$ .

### Tensor Quadrilateral Prism

In the same way, we can produce a prism, identical in point-set topology to the hexahedron, from the tensor product of a quadrilateral and a segment, shown in Tables 1.16 and 1.17. We can describe the group as being generated by a rotation of  $90^\circ$  about the  $z$ -axis (a), an exchange of the top and bottom faces (b), and a reflection (r). We again number the orientations such that the reflection of orientation  $o$  is  $-(o + 1)$ .

### Orienting Separating Faces

In Fig. 1.4,  $p = 0$  is a cell with height 0, cone point 1 ( $c = 1$ ) is an edge  $q = 7$  with height 1, and  $\text{cone}(q) = \{3, 4\}$  which are vertices of height 2, or depth 0. There can only be one sorted order for the edge 7, which here is  $\{3, 4\}$ , but for cell 1 we want the order to be  $\{4, 3\}$ . We attach an orientation to the arrow ( $p, q$ ) in the DAG indicating that  $q$  should be reordered when used in cell  $p$ . The orientation indicates which group element from the dihedral group of  $q$  should transform the stored order to the order needed in cell  $p$ . For an edge, there are only two transformations, the identity (0) and flipping (-1).

Orientation	<b>0</b>	<b>1</b>
Arrangement		
Orientation	<b>-1</b>	<b>-2</b>
Arrangement		
<hr/>		
Orientation	<b>2</b>	<b>3</b>
Arrangement		
Orientation	<b>-3</b>	<b>-4</b>
Arrangement		

Table 1.13: The symmetry group for the square pyramid

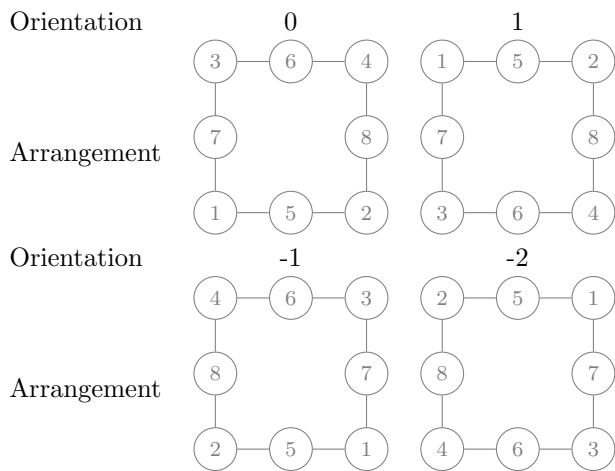


Table 1.14: The symmetry group for the tensor product of two segments

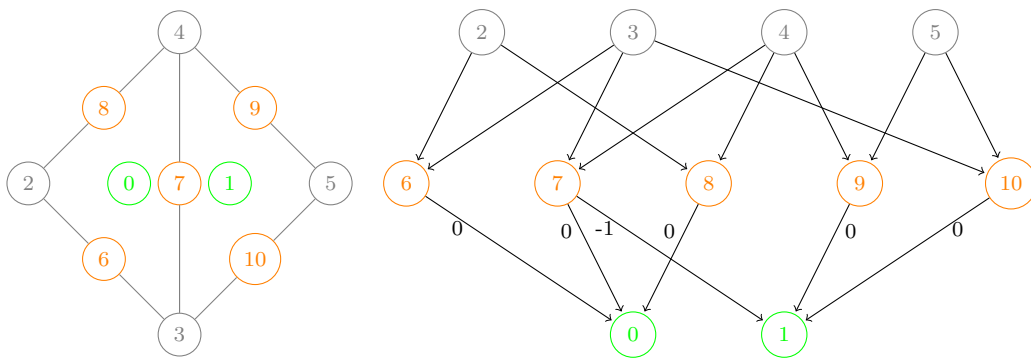


Figure 1.4: Two triangles sharing a common edge, and the corresponding Hasse digram.



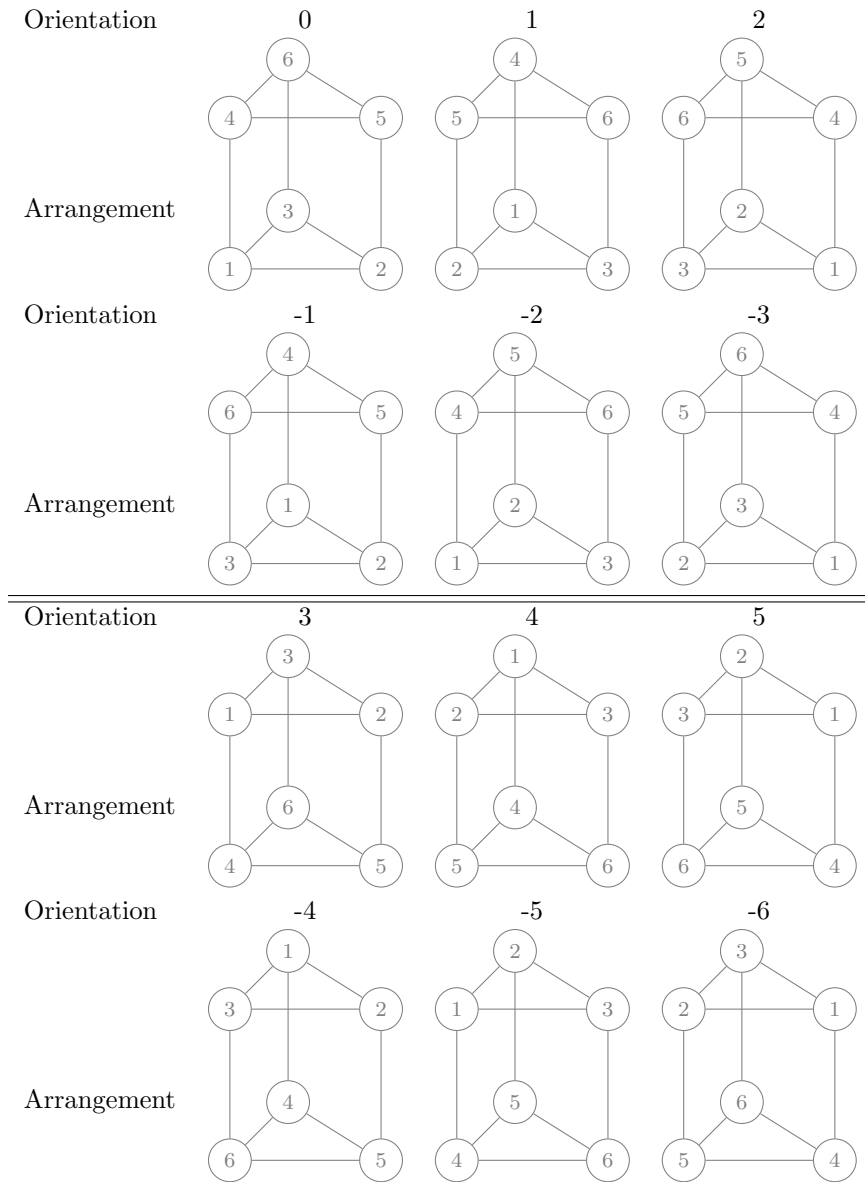


Table 1.15: The symmetry group for the tensor product of a triangle and a segments

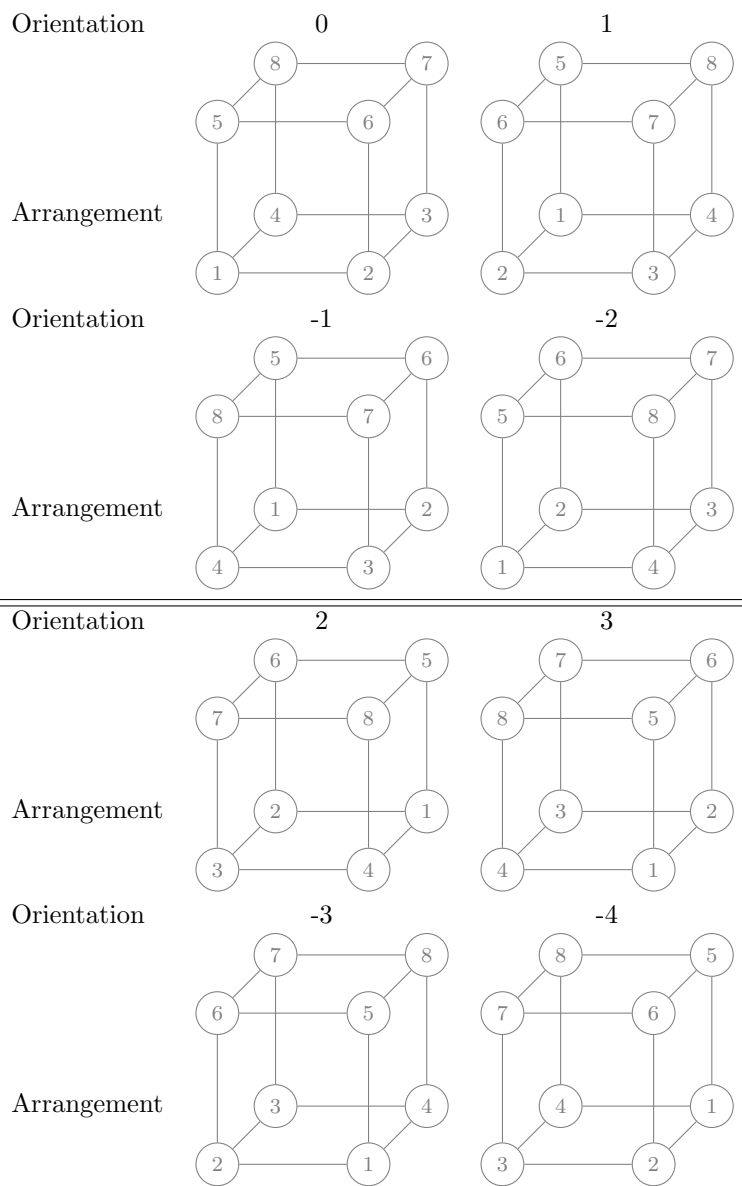
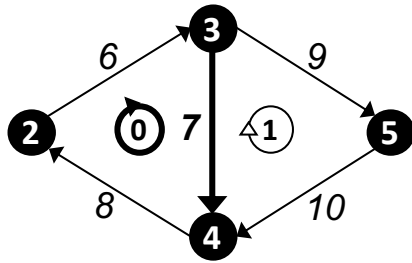
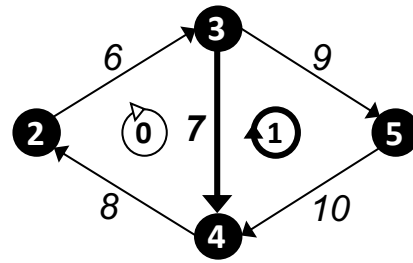


Table 1.16: Part I: The symmetry group for the tensor product of a quadrilateral and a segments

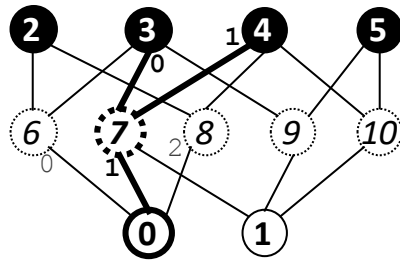




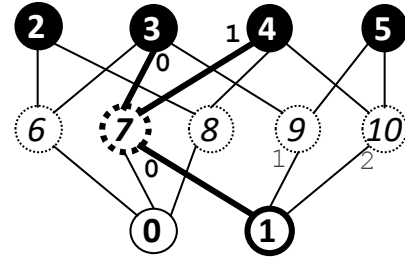
(a) edge 7 in cone of face 0,  
 $C(0, 1) = 7$



(b) edge 7 in cone of face 1,  
 $C(1, 0) = 7$



(c) starting point of edge 7 within face 0 is 3,  
 $C(C(0, 1), S(0, 1)) = C(7, 0) = 3$ ,  
 $O(0, 1) = 0$



(d) starting point of edge 7 within face 1 is 4,  
 $C(C(1, 0), S(1, 0)) = C(7, 1) = 4$   
 $O(1, 0) = -2$

Figure 1.5: Mesh from Figs. ?? with cone points order and orientation. We focus here on the edge 7 within the cones of faces 0 and 1.

### 1.1.4 Periodicity

### 1.1.5 CAD Interface

## 1.2 Nonconforming Topology

### 1.2.1 The Parent Tree

### 1.2.2 Anchors and Constraints

## 1.3 Submeshes

DMLabel - Hash table + Sorted list - Automatic conversion, good for batches of queries and updates

## 1.4 Parallelism

A parallel Plex is nothing more than a collection of serial Plexes along with a mapping identifying points between them. For this mapping, we use the PetscSF object. It relates things on different process which can be numbered with PetscInt, but may have different numbers on different processes. Thus, the SF can relate point 5 on the current process with point 27 on process  $p$ . This mapping allows an arbitrary overlap for parallel meshes, and means that PetscSF can automatically compute the communication pattern and create the MPI structures for all mesh communication.

The guarantee that Plex makes in parallel is that the closure of any point  $p$  will always be present locally. This allows all queries based upon cell shapes to succeed in parallel without extra communication and marshalling. We make this choice because it is common to think of dividing up space into pieces with well-defined shapes, and the combinatorial topology is usually bases on a definition of the boundary operator, closely related to our closure operation. Moreover, we will base our description of mesh symmetries on symmetries of cell closures. Alternatively, we could guarantee that the star of any point is present locally, and consider the symmetries of the edges around a vertex, or the cells around a face, rather than the faces around a cell. However, this seems less natural for most numerical methods.

Note that it is not possible for both the closure and star to be present locally and for the mesh to be consistent in parallel. We must choose one operation to preserve. For example, if we add a cell from another process in order to complete the star of a face, then that cell will initially be missing a portion of its closure. If we add this closure, we can introduce points on its boundary that are missing parts of their star.

### 1.4.1 Distribution

Bootstrap from process graph

## References

- Alexandroff, Paul (1932). *Elementary concepts of topology*. translated by Alan Farley (Dover 1960).
- Dimension theory* (June 2021). Encyclopedia of Mathematics. URL: [7http://encyclopediaofmath.org/index.php?title=Dimension\\_theory&oldid=46706%7D](http://encyclopediaofmath.org/index.php?title=Dimension_theory&oldid=46706).

# Chapter 2

# Functions

*The purpose of computing is insight, not numbers.*

— Richard Hamming

## 2.1 PetscSection

**Start with section from PETSc manual** The purpose of a PetscSection is to determine a data layout over the mesh by associating degrees-of-freedom, *dof*, to each mesh point.

Explain local and global section.

### 2.1.1 Constraints

Constraints are stored in a subsection and IS.

### 2.1.2 Symmetries

If dofs are associated to a mesh point that has a symmetry group, such as  $S_2$  for the segment, then the ordering of dofs should change when the arrangement of the  $k$ -cell changes. We need any arrangement of the cell to also define a valid data layout. An example will help illustrate this principle. If we have a  $P_3$  element on a triangle, or a  $Q_3$  element on a quadrilateral, then two dofs are associated with each edge. However, we think of each dof as being located on a "node" on the edge, and these nodes divide the edge into thirds. This means that I have to know which value is associated with which node, not just the edge itself. Note here that these nodes correspond exactly to point evaluation functionals in the dual basis. The dofs must have some ordering in the local section, and we will assume that it follows the canonical arrangement of the mesh point. If the mesh point is attached to a cell in another arrangement, using an orientation different from zero, then that cell should see the dofs in an order that matches the orientation. In our example, shown in Fig. 2.1, the

Draw a picture of two quad cells, showing the dof layout.

Figure 2.1: Dofs on a pair of  $Q_3$  cells

left cell sees the edge in the canonical ordering, and thus the dofs as  $\{24, 25\}$ , whereas the right cell sees the edge reversed, in orientation  $-1$ , so that the dofs are ordered  $\{25, 24\}$ .

In general, if the cell transforms with group  $G$  and has  $n$  dofs, then the dofs should transform as an  $n$ -dimensional representation of the group  $G$ . There is a complication if the dual basis vector has non-trivial transformation behavior. For example, if the functional is a vector, then it will acquire a minus sign under inversion, whereas scalar dofs will not.

Toby computes the transformation of the dual basis, given the transformation of vertices from the orientation. This is fine to preserve.

We need a way to speed up the lookup. I think it is to store representations of the symmetry groups for known cell types, and use that memory directly with an integer tag in the Section rather than a DMLabel lookup. This does trade more storage for speed, but this is usually a good trade with Section. We can omit the storage if no cells requires a non-trivial permutation.

## 2.2 Parallelism

### 2.2.1 Local and Global Sections

### 2.2.2 Data Distribution

## 2.3 Periodicity

## 2.4 Global Basis Transformation

## 2.5 Geometry

## 2.6 Projection

The projection functionality of Plex is intended to make it easy for users to define useful projection operations that commonly occur in numerical code. Although projection into some space will always be the last step, we do not confine ourselves to pure projections, but rather allow some preprocessing in order to simplify the procedure.

### 2.6.1 Interpolation

Interpolation is perhaps the most common type of projection. We can easily see the interpolation is a projection, since if the input lies in the space, the action



is just the identity, meaning that  $\mathcal{I}^2 = \mathcal{I}$ . By interpolation, we will mean that the action of the dual basis vectors on the function is the same as the action on the interpolant. To see that this makes sense, let's take the example of a dual basis biorthogonal to the primal basis,

$$\phi_i^\dagger \phi_j = \delta_{ij}. \quad (2.1)$$

Now the action of the dual basis vector gives the coefficient,

$$\phi_i^\dagger v = \phi_i^\dagger \sum_j v_j \phi_j \quad (2.2)$$

$$= \sum_j v_j \phi_i^\dagger \phi_j \quad (2.3)$$

$$= \sum_j v_j \delta_{ij} \quad (2.4)$$

$$= v_i \quad (2.5)$$

so if we set the coefficients to this action

$$u_i^* = \phi_i^\dagger u, \quad (2.6)$$

then interpolation will be idempotent for functions in the space.

Now we will let the input function come from a different finite element space, and perhaps even a different mesh. Each dual basis vector can be expressed as a quadrature rule, as a consequence of the Riesz-Markov-Kakutani representation theorem (Wikipedia 2015). The quadrature points are guaranteed to lie in the cell closure, so we need only evaluate the source basis on the reference quadrature points if source and target spaces share a mesh. On non-matching meshes, we would have to locate the quadrature points in the source mesh and then evaluate the function at those points.

### 2.6.2 $L_2$ projection

If we minimize the  $L_2$  norm between a function and its representer in the target space, we may write this as

$$\mathcal{P}u \equiv u^* = \min_{v \in V} \frac{1}{2} \|u - v\|_2^2. \quad (2.7)$$

The first order conditions for this problem are

$$\int \phi u = \int \phi u^* \quad \forall \phi \in V, \quad (2.8)$$

which we can also interpret as the requirement that  $u$  and  $u^*$  are *weakly equivalent*, meaning that they share the same moments for all functions in the target

space. If we expand  $u^*$  in this basis,

$$\int \phi_i \sum_j u_j^* \phi_j = \int \phi_i u \quad (2.9)$$

$$M\mathbf{u}^* = \mathbf{u} \quad (2.10)$$

where  $M$  is the mass matrix, and  $\mathbf{u}$  is a vector of the moments of  $u$ .

We can implement  $L_2$  projection by giving the identity kernel for the  $g_0$  Jacobian (producing the mass matrix), and function  $u$  for the  $f_0$  residual kernel. However, we can consider that case that  $u$  is also a finite element function, but in a different space on the same mesh or on a different mesh. If  $u$  is on the same mesh, we need only tabulate that basis once on the new quadrature points since the cells match. If the meshes are different, we will have to locate each set of quadrature in the source mesh, which argues for batching groups of cells, and interpolate them separately, as discussed above.

## References

Wikipedia (2015). *Riesz-Markov-Kakutani Representation Theorem*. [http://en.wikipedia.org/wiki/Riesz-Markov-Kakutani\\_representation\\_theorem](http://en.wikipedia.org/wiki/Riesz-Markov-Kakutani_representation_theorem). URL: [http://en.wikipedia.org/wiki/Riesz-Markov-Kakutani\\_representation\\_theorem](http://en.wikipedia.org/wiki/Riesz-Markov-Kakutani_representation_theorem).

**Part II**

**Transformations**



## Chapter 3

# General Transformations

*As Gregor Samsa awoke one morning from uneasy dreams, he found himself transformed in his bed into a gigantic insect.*

— Franz Kafka

Having learned to represent computational meshes in Part I, we can now think about changing a given mesh into another. For example, we might want to take in a mesh and deliver back a refined version of it, or perhaps a coarsened version, or do this adaptively. We can imagine receiving only cells and vertices, and creating the associated faces and edges automatically. We could transform all cells in the mesh to simplices, or to box cells. We could take in a surface mesh, and extrude it in the normal direction to create a volumetric mesh, or do this only in part of the mesh to produce a refined boundary layer. Given two meshes, we could produce a common refinement of them both.

Our goal in this chapter will be to identify, out of the myriad possible mesh transformations, those that are efficiently computable, considering both the time and space complexity. We would like a general strategy for producing the transformed mesh, given an input mesh, which is valid also in parallel. Moreover, we will strive to develop algorithms whose complexity is *output sensitive*. By this we mean that the cost is proportional to the fraction of the transformed mesh that we choose to output. With such an algorithm, we could envision producing the transformed mesh “on the fly”, so that it need not be stored but the needed portions could be computing on demand.

Suppose that a point  $p$  in the input mesh produces a point  $q$  in the transformed mesh. We would like a kind of locality, meaning that the cone of  $q$  was easily discoverable given  $p$ . The simplest rule of this kind would be that the cone of  $p$  produces the cone of  $q$ . However, this rules out many of our cases above, such as regular refinement of a mesh. Thus we will begin with a slightly more expansive rule,

**Condition 1** *The cone of a point  $q$  of the transformed mesh is produced by the closure of a point  $p$  in the input mesh.*

which we can write

$$\text{cone}(\text{child}(p)) \in \text{child}(\text{cl}(p)), \quad (3.1)$$

where we use  $\text{child}(p)$  to indicate the set of points produced by point  $p$ , or the “production cone” of  $p$ . There will be a similar dual notion, a “production support”, which we will call  $\text{parent}(q)$  meaning the set of points which can produce  $q$ . With this condition, we need only compute the part of the transformed mesh produced by the closure of  $p$  in order to capture the cone of  $q$ . This gives us an easy way to bound both computation and storage costs for our virtual mesh.

What about the closure of  $q$ ? Consider a point  $q'$  in the cone of  $q$ , so that

$$q' \in \text{cone}(q) \quad (3.2)$$

$$\in \text{child}(\text{cl}(p)) \quad (3.3)$$

by Condition 1. This means there is some point  $p'$  in the closure of  $p$  that produces  $q'$ . Thus,

$$\text{cone}(q') \in \text{cone}(\text{child}(p')) \quad (3.4)$$

$$\in \text{child}(\text{cl}(p')) \quad (3.5)$$

$$\in \text{child}(\text{cl}(p)) \quad (3.6)$$

where the last line follows because transitive closures are nested. By reasoning this way for each point in the closure, we can conclude that

$$\text{cl}(\text{child}(p)) \in \text{child}(\text{cl}(p)). \quad (3.7)$$

We have now bounded the storage requirements for our transformed mesh, but how costly is it to identify the produced points? In concrete terms, how can one compute a total order on the points in the transformed mesh? Consider the situation for mesh interpolation. We can think of each cell producing the faces and edges in its closure. However, this would mean that multiple cells would produce the same face or edge. To compute a numbering, we have to compute a signature for each point introduced, say its vertex cone, and then compare signatures to establish identity. In order to limit the resources for this comparison, we need some limit on the parent set for a point  $q$ . The simplest condition is

**Condition 2** *A point  $q$  of the transformed mesh is produced by only one  $p$  in the input mesh, such that  $|\text{parent}(q)| = 1$ .*

This rule allows a unique numbering to be calculated given only a prior numbering of the producing Plex. Even in parallel, local numbering can be calculated independently and patched together using PetscSF. Using Condition 2, we can now bound the cost of support queries. Let a point  $q$  be produced by a point  $p$ , and consider a point  $q'$  in the star of  $q$ ,

$$q' \in \text{st}(q) \iff q \in \text{cl}(q')$$

from Eq. 1.3. Let  $q'$  be produced by a point  $p'$ ,

$$q' \in \text{child}(p') \quad (3.8)$$

so that

$$\text{cl}(q') \in \text{child}(\text{cl}(p')) \quad (3.9)$$

$$q \in \text{child}(\text{cl}(p')) \quad (3.10)$$

We can now use that fact that parents are unique, Condition 2,

$$\text{parent}(q) \in \text{cl}(p') \quad (3.11)$$

$$p \in \text{cl}(p') \quad (3.12)$$

$$p' \in \text{st}(p) \quad (3.13)$$

so that we have shown

$$\text{st}(\text{child}(p)) \in \text{child}(\text{st}(p)). \quad (3.14)$$

Thus, to compute a star for any point in the transformed mesh, we need only consider the star of the parent in the input mesh.

A more sophisticated condition would bound the set of possible parents. For example, we could require that

$$\text{parent}(\text{st}(q)) \in \text{st}(\text{parent}(q)) \quad (3.15)$$

which would produce the same kind of locality. This is satisfied by the interpolation algorithm, even though parents are not unique. In parallel, this could become problematic as regions to check spread across process boundaries. Interpolation, fortunately, has another property. All the produced points that are shared are guaranteed to appear in the SF, which we will see in detail in Chapter 4.

### 3.1 Definition

A transformation will be defined by its action on each cell, in that for each cell in the source mesh the transformation will produce a set of cells in the target mesh. In the simplest examples, such as regular refinement, the action depends only on the cell type. However, we will allow the transformation to make different decisions for cells of the same type. In the implementation, this will be accomplished using a label to differentiate the cells, giving each cell a *transformation type* to refine its cell type. Thus, in the discussion below we will refer to transformation type, but the reader can imagine this as a stand-in for the cell type in order to get an intuitive feel for the algorithm. As we discuss the definition below, we will use regular tetrahedral refinement as a non-trivial example to illustrate the stages.

In our definition, we first indicate which cell types are produced for a given transformation type. This allows the stratification of the target mesh to be easily computed, and also separates the different transformation rules, simplifying the description. In our tetrahedral refinement example, we need to consider the transformation of four cell types: vertices, edges, triangular faces, and tetrahedral cells. The vertices produce identical copies, and thus have a single production type POINT. Edges are split into two pieces, yielding one POINT in the center, and two SEGMENTS. The triangular faces are divided into four, producing three SEGMENTS and four TRIANGLES. Notice that the subdivided triangles do not count the edges and vertices introduced on the boundary because those are handled by the edge transformation rule. In general, the transformation rule only deals with the interior, not the boundary of a cell. Finally, the tetrahedron is divided into eight, producing one SEGMENT, eight TRIANGLES, and eight TETRAHEDRA. Since several points can be produced with the same cell type, we will number them using a *replica number*, which we denote as  $r$ .

In order to describe the cells which are produced, we need to know their cones and cone orientations. The cones consist of points in the target mesh, so we must have some way of identifying them. If we just needed to refer to points produced by the current point, we could use the celltype and replica number. However, this will not in general be sufficient. Instead we make use of Condition 1, which says that the cone of any point produced must be contained in the set of points produced from the closure of the original point. Thus, we need to first locate a point in the closure of the original point, and then specify the target point produced from it. In the code, this information is returned by `DMPlexTransformCellTransform()`.

Let us first consider the lone segment produced by dividing the tetrahedron, shown in Fig. 3.1. We can describe the cone of this segment with the following array

---

```
{DM_POLYTOPE_POINT, 2, 0, 0, 0, 0, DM_POLYTOPE_POINT, 2, 2, 1, 0};
```

---

The first cone point is a vertex, and we arrive at the producing point by taking two cones. First, we take cone point 0 of the tetrahedron, which is the bottom face. Then we take cone point 0 in that bottom face, which is the left edge. Finally, we take the first vertex produced by that edge, which is replica 0. In the reference cell, this vertex has coordinates  $(-1, 0, -1)$ . We can carry out the same computation for the second cone point, which also requires taking two cones. We take cone point 2 of the tetrahedron, which is the front face, and then cone point 1 of that face, which is the diagonal edge, and finally the first vertex produced by that edge, which has coordinates  $(0, -1, 0)$ . To complete the definition, we should also specify the orientation of each cone point, but vertices can only have orientation 0. Notice that the description of these points is not unique. For example, we could have used face 3 to extract the second vertex.

We proceed in the same manner for all target points produced from this source point. The first internal triangular subface can be described using

---



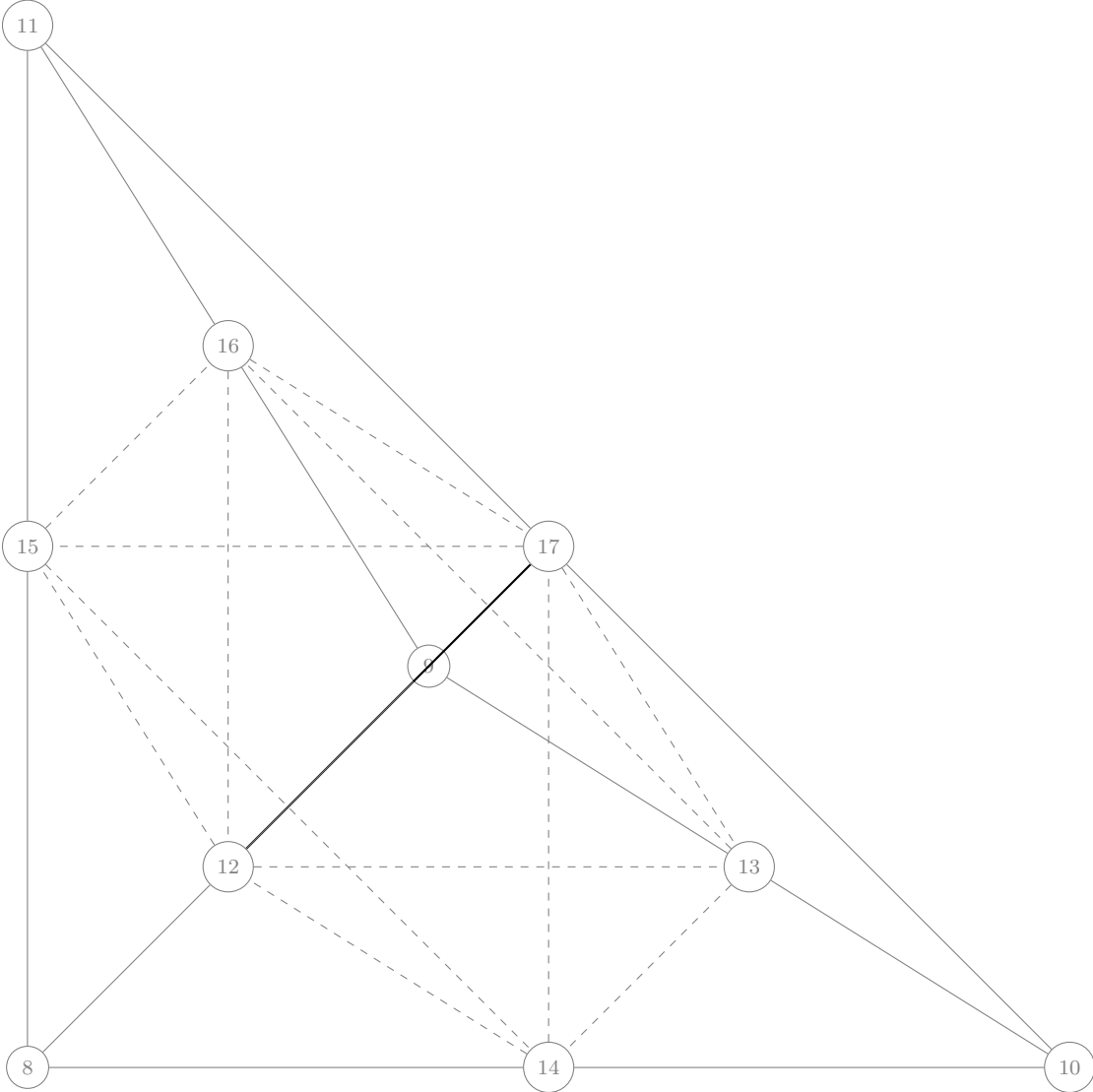


Figure 3.1: A single regular refinement of a tetrahedron. We have indicated the interior segment in black, and used dashed lines for the segments produced by the boundary faces.

---

```
{DM_POLYTOPE_SEGMENT, 1, 0, 2,
DM_POLYTOPE_SEGMENT, 1, 1, 2,
DM_POLYTOPE_SEGMENT, 1, 2, 2};
```

---

The first edge is produced from a point in the cone of the tetrahedron, so we only need 1 cone operation. It comes from face 0 of the tetrahedron, which is the bottom face, and is the third segment produced, which means it connects the left edge with the front edge. The second edge is produced from face 1, which is the left face, and is the third segment produced, meaning it connects the bottom edge (left edge from before) with the vertical edge. Finally, the third edge is produced from face 2, which is the front face, and is the third segment produced, which connects the vertical edge to the bottom edge (front edge from the first face). Thus we have a triangle, and the edges are properly oriented. On other internal triangles, for example

---

```
{DM_POLYTOPE_SEGMENT, 1, 0, 2,
DM_POLYTOPE_SEGMENT, 0, 0,
DM_POLYTOPE_SEGMENT, 1, 2, 0};
```

---

some segments need to be reversed, in this case the last one. Therefore we must also provide an orientation array, which here would look like

---

```
{0, 0, -1};
```

---

Note also that the middle segment is produced directly by the tetrahedron, and thus 0 cones need to be taken.

If we look at the first subtetrahedron, we have the expression

---

```
{DM_POLYTOPE_TRIANGLE, 1, 0, 0, DM_POLYTOPE_TRIANGLE, 1, 1, 0,
DM_POLYTOPE_TRIANGLE, 1, 2, 0, DM_POLYTOPE_TRIANGLE, 0, 0};
```

---

The first face is produced from face 0 in the tetrahedron, which is the bottom face. It is the first triangle produced, meaning it is the triangle containing the lower-left vertex. The second face is produced from face 1, and is also the triangle containing the lower-left vertex. Similarly, the third face comes from face 2 of the tetrahedron, and is the triangle containing the lower-left vertex. Finally, the last face is produced directly by the tetrahedron itself, and is the first such face, which divides the lower-left vertex,  $(-1, -1, -1)$ , from the rest of the tetrahedron. Also, each of these faces is properly oriented. However, this will not be true of all the subtetrahedra, so we will need a way of discovering the orientations automatically, detailed in Section 7.1.

## 3.2 Group Action

Our definition is able to give us the cone for any point produced from a cell in orientation 0. However, what happens when our original point has a nonzero orientation? Since orientations are symmetry transformations of the cell, it

must be the case that an identical cell type is produced. However, the cell that is produced might correspond to a different replica number for that type, and have a different orientation than originally specified. Hence, we can think of our group action  $\mathcal{G}$  as mapping, for any cell type  $ct$  produced,

$$\mathcal{G}(ct, r, o) \longrightarrow (ct, r', o').$$

We can generate this action automatically by running our transformation on the reference cell, and comparing the cones from points that are produced. We will cover the implementation of this discovery in Section 7.1.

### 3.3 Numbering

The defining feature of our definition for a mesh transformation is that the transformed mesh can be known efficiently purely from the definition of the input mesh and the transformation. The first step is to introduce a map from source points, those in the original mesh, to target points, those in the transformed mesh, and also its inverse. We will think of this numbering as an index structure, built on top of the transformation mechanism discussed above. We will first describe the situation when the cell type determines the transformation, and then extend this to the situation when we have many transformation types.

Suppose that we are given a source point  $p$  and its cell type  $ct_p$ , a cell type that is being produced  $ct_q$ , and a replica number  $r$ , then we can compute the number of the produced point  $q$  in the transformed mesh. During the setup phase, we precompute data needed for the index. First, a total order on cell types. This might not match the enumeration value since we often want to number cells, then vertices, then faces, and then edges so that interpolation does not affect the ordering. Next, we run over the original mesh, calling `DMPlexTransformCellTransform()` for each point, and record the outputs. This allows us to calculate offsets for each celltype in the transformed mesh. In fact, we calculate the offset for the first cell of given target cell type produced by a cell of given source cell type. Thus given the source cell type, we can lookup the offset for the output cell type, which we will call  $\text{off}(ct_p, ct_q)$ . Next we look at the cell transform description for the given cell type. We can see the number of replicas of the target cell type for each source point, called  $N_r(ct_p)$ , which we multiply by the *reduced point number*  $rp$ , meaning that the source point is the  $rp$ th point of the source cell type. In total, our target point number is

$$q = \text{off} + rp * N_r + r.$$

The only difference when allowing different transformation types for a given cell type, is that the offset is calculated using the transformation type, and the reduced point number is computed with respect to the transformation type.

### 3.4 Implementation

If we assume the uniqueness condition 2, we shall show that the resulting mesh need not be stored, since we can build queries on the parent mesh. In order to demonstrate this, we must first consider how the basic mesh queries are implemented for a Plex. For the basic implementation, we explicitly store both the cone and support of each mesh point. The transitive closure is built from repeated calls to the cone and support routines, while discarding duplicate points. A similar process is used for the implementation of the meet and join operations. Thus, if we could produce the cone and support of each point in the transformed mesh, we could use our prior implementation to generate all queries for it.

The formation of the cone of produced points is more straightforward, and we will address it first. We must know the size of the cone for any produced point, but we can discover its type by searching the index structure for point numbering. The celltype determines the size of the cone, and in fact more. Using `DMPlexTransformGetSourcePoint()` we can also recover the producing point  $p$ , its celltype, and the replica number  $r$  for the new point. The `DMPlexTransformGetTargetPoint()` is the inverse, giving us the number of the produced point given information about the producing point. Once we have the producing, or parent, point  $p$ , we can march through its cone, using the data structure from Section 3.1 to discover each point produced in the cone, its orientation based on the group action for the producing point, and then get its number in the transformed mesh. The code for this, take from `DMPlexTransformGetCone()`, is shown below.

---

```

DMPlexGetCone(dm, p, &cone);
for (c = 0; c < csizeNew; ++c) {
    /* Parent Parent point: Parent of point pp */
    PetscInt ppp = -1;
    /* Parent point: Point in the original mesh producing new cone point */
    PetscInt pp = p;
    /* Orientation of parent point pp in parent parent point ppp */
    PetscInt po = 0;
    /* Parent type: Cell type for parent of new cone point */
    DMPolytopeType pct = ct;
    /* Parent cone: Cone of parent point pp */
    const PetscInt *pcone = cone;
    /* Replica number of pp that produces new cone point */
    PetscInt pr = -1;
    /* Cell type for new cone point of pNew */
    const DMPolytopeType ft = rcone[coff++];
    /* Number of cones of p that need to be taken when producing new cone point */
    const PetscInt fn = rcone[coff++];
    /* Orientation of new cone point in pNew */
    PetscInt fo = rornt[ooff++];
    PetscInt lc;

```

```

/* Get the type (pct) and point number (pp) of the
   parent point in the original mesh which produces this cone point */
for (lc = 0; lc < fn; ++lc) {
    const PetscInt *parr = DMPolytopeTypeGetArrangment(pct, po);
    const PetscInt acp = rcone[coff++];
    const PetscInt pcp = parr[acp*2];
    const PetscInt pco = parr[acp*2+1];
    const PetscInt *ppornt;

    ppp = pp;
    pp = pcone[pcp];
    DMPlexGetCellType(dm, pp, &pct);
    DMPlexGetCone(dm, pp, &pcone);
    DMPlexGetConeOrientation(dm, ppp, &ppornt);
    po = DMPolytopeTypeComposeOrientation(pct, ppornt[pcp], pco);
}
pr = rcone[coff++];
/* Orientation po of pp maps (pr, fo) -> (pr', fo') */
DMPlexTransformGetSubcellOrientation(tr, pct, pp, fn ? po : o, ft, pr, fo, &pr, &fo);
DMPlexTransformGetTargetPoint(tr, pct, ft, pp, pr, &coneNew[c]);
orntNew[c] = fo;
}

```

---

The `rcone[]` array and `coff` offset are the information encoding the cones of produced points, while `rornt[]` and `ooff` encode orientation of the cone points. The `DMPolytopeTypeGetArrangment()` function gives the permutation of the cone corresponding to a given orientation for the input celltype.

In order to produce the support of a point in the transformed mesh, we must work a little harder. Since we have the relation (3.14), we know that the star of a produced point  $q$  is contained in the produced points of the star of the parent point  $p$ . Thus, we could construct explicitly the patch of the transformed mesh produced from  $st(p)$ , and then run our support query as usual. If many support queries are desired, then an index structure should be built to defray some of this cost.



## Chapter 4

# Interpolating

*The purpose of computing is insight, not numbers.*

— Richard Hamming

Use quote from my small Topology book.

Topological *interpolation* is the process of constructing intermediate levels of the ranked poset describing a mesh, given information at bracketing levels. For example, if we receive triangles and their covering vertices, as in Fig. 4.1, interpolation will construct the edges. The first algorithm for interpolation on the Hasse diagram was published in (Logg 2009), but this version was only appropriate for simplices, ignores orientation of the mesh points, and did not give a complexity bound. The version below rectifies these shortcomings, and was first described in (Hapla et al. 2021).

The interpolation procedure selects a given point stratum as cells, for which it will construct faces. It iterates over the cells, whose cones are oriented sets of vertices. The two essential operations are to extract an oriented face from the vertex set, and then attach it to a cell with the correct orientation. Orientation of mesh points is detailed in Sec. 1.1.3. In order to enumerate the faces for a given cell type, we have `DMPlexGetFaces_Internal()` for homogeneous cells, and `DMPlexGetRawFacesHybrid_Internal()` for *hybrid* cells, meaning cells with more than one face type, such as prisms. These functions return oriented lists of vertices for each face of the input cell.

### 4.1 Serial Algorithm

An initial iteration over cells construct all faces, and enters them into a hash table, where the hash key is the sorted list of vertices in each face. When dealing with hybrid cells, we need one pass for each type of face. Once the hash table is constructed, we know the number of new faces to be inserted, and can allocate a new Plex. This Plex is identical to the old, except that it has a new face

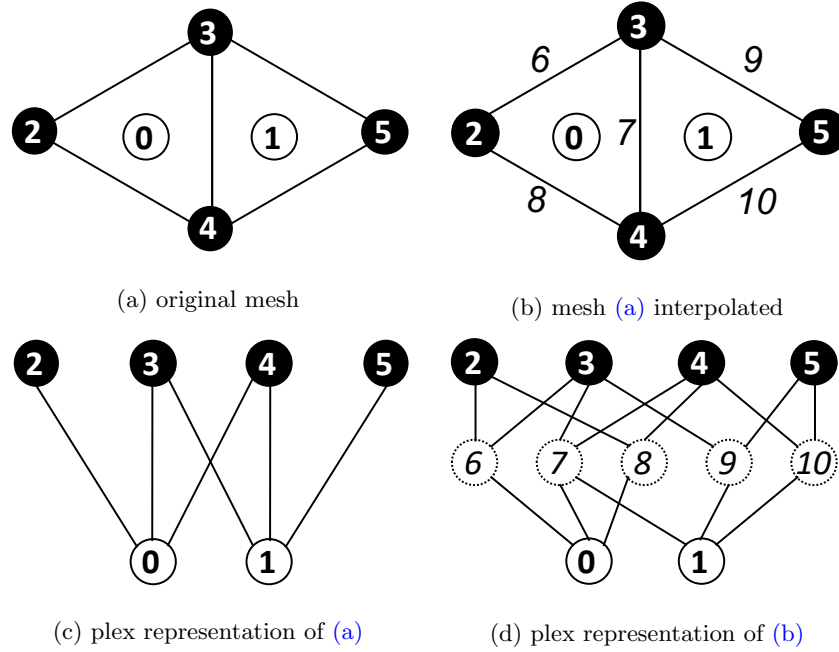


Figure 4.1: Sequential topological interpolation.

stratum, and the cone sizes of cells may have changed. For example, hexahedral cells have 8 vertices, but 6 faces, so that cone size would change from 8 to 6.

A second iteration over cells inserts the faces. We clear the hash table and repeat the face extraction above. If a face is missing from the table, we insert it into the table, record its cone, and also insert it into the cell cone with default orientation. If instead it is present in the table, we insert it into the cell cone with orientation computed from `DMPlexGetFaces_Internal()`. Again, for hybrid cells, we need one pass for each face type.

In order to interpolate an entire mesh, we loop over cell stratum from height zero, the highest dimensional points, to depth one, the edges. For example, for a hexahedral mesh, we would interpolate quadrilateral faces, and then edges. The complexity to interpolate a given stratum is in  $\mathcal{O}(2F_C N_C N_H)$ , where  $N_C$  is the number of cells,  $F_C$  is the number of faces per cell, and  $N_H$  is the number of face types per cell. The leading term for a hexahedral mesh is  $16N_0 + 8N_1 + 4N_2 = 16C + 8F + 4E$ , where  $C$  is the number of cells,  $F$  the number of quadrilateral faces, and  $E$  the number of edges. Clearly, the operations are all linear in the mesh size.



## 4.2 Parallel algorithm

If interpolation is performed on a parallel mesh, the first step consists in applying the sequential topological interpolation (Sec. 4.1) and cone orientation (Sec. 1.1.3) on each rank independently.

Then we must alter the `PetscSF` structure which identifies mesh points which are owned by different processes, or *leaf* points. The SF structure is described in the PETSc manual and (Zhang et al. 2022). We mark all leaf points which are adjacent to another ghost point as candidates. These candidate points are then gathered to *root* point owners (using `PetscSFbcst()`). For each candidate, for each point in the cone, the root checks that either it owns that point in the SF or it is a local point. If so, it claims ownership. These claims are again broadcast, allowing a new SF to be created incorporating the new edges/faces.

However, the cone orientation has been done on each rank independently, meaning that it is only partition-wise correct. In order to make these consistent, we use the fact that interface edges/faces owned by different ranks represent the same geometrical entity, i.e. they are connected by the `pointSF`, like edges  $5_0$  and  $6_1$  in Fig. 4.2, then they must have a conforming order of cone points ( $p_r$  means that point  $p$  is owned by rank  $r$ ). If we let  $p \rightarrow q$  denote that  $p$  is a leaf of root  $q$  in the `pointSF`, then we can write this requirement as an implication

$$\left. \begin{array}{l} p_0 \quad \rightarrow p_1 \\ \text{cone}(p_0) = (q_0^0, \dots, q_0^{n-1}) \\ \text{cone}(p_1) = (q_1^0, \dots, q_1^{n-1}) \end{array} \right\} \Rightarrow \left\{ \begin{array}{l} q_0^0 \quad \rightarrow q_1^0 \\ \dots \\ q_0^{n-1} \quad \rightarrow q_1^{n-1}. \end{array} \right. \quad (4.1)$$

However, if this implication is not satisfied, it must be true for some permutation in the dihedral group of  $p$ , and our job is to find this permutation, which will give us the new orientation of the point  $p_1$ .

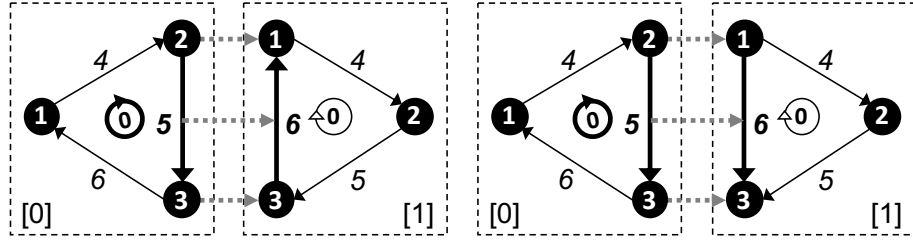
In Fig. 4.2 the implication is violated for the edges  $5_0$  and  $6_1$ . They are flipped with respect to each other, or more formally the `pointSF` connects the edge and its incident vertices

$$5_0 \rightarrow 6_1, \quad 2_0 \rightarrow 1_1, \quad 3_0 \rightarrow 3_1,$$

but the order of cone points does not conform,

$$\begin{aligned} 2_0 = \text{cone}(5_0)[0] &\not\rightarrow \text{cone}(6_1)[0] = 3_1, \\ 3_0 = \text{cone}(5_0)[1] &\not\rightarrow \text{cone}(6_1)[1] = 1_1. \end{aligned}$$

and only inverting permutations can bring them into coincidence, and would lead to an incorrect PDE solution if the discretization made use of the edge. In order to satisfy this requirement, and additional synchronization of the interface cones must be carried out. We start by synchronization of the interface cone point numbering. Remember that the `pointSF` is a one-sided structure, so only the origins of the arrows can be found directly. Let us assume an edge/face  $p$  on rank  $r$ ,  $p_r$ , and a `pointSF` arrow pointing from  $p_r$  to some  $p_s$ ,  $p_r \rightarrow p_s$ . If



(a) mesh from Fig. 4.1 with non-conforming cone orientation

(b) mesh from Fig. 4.1 with conforming cone orientation

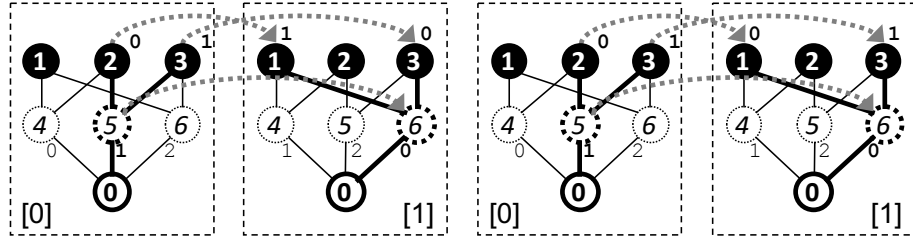
(c) plex representation of (a);  $\text{ornt}(0_1)[0] = 0$ (d) plex representation of (b);  $\text{ornt}(0_1)[0] = -2$ 

Figure 4.2: Parallel DMPlex with cone points order and orientation.

we detect an arrow directed from the cone point  $q_r = \text{cone}(p_r)[c]$  to  $q_s$ , we set  $\text{root}(q_r) = q_s$ , otherwise  $\text{root}(q_r) = q_r$ . This  $\text{root}(q_r)$  is sent to  $s$  using `PetscSFBCastBegin/End()`, and stored at the destination rank as  $\text{leaf}(q_s)$ . This is done for each rank, each point with height greater than zero, and each cone point.

Now from the rank  $s$  viewpoint, for the cone of point  $p_s$ , it has  $\text{root}(\text{cone}(p_s))$  and the received  $\text{leaf}(\text{cone}(p_s))$ . If  $\text{root}(q_s) = \text{leaf}(q_s)$  does not hold for all cone points, we must transform the cone so that this condition is satisfied. We do this by updating the orientation  $\text{ornt}(t)$  for all points  $t \in \text{st}(p_s)$  accordingly to compensate for the change of cone order. We can see that orientation synchronization relies heavily on the `pointSF`, which is why it must be processed first.

## References

- Logg, Anders (2009). “Efficient representation of computational meshes”. In: *International Journal of Computational Science and Engineering* 4.4, pp. 283–295.
- Hapla, Vaclav, Matthew G. Knepley, Michael Afanasiev, Christian Boehm, Martin van Driel, Lion Krischer, and Andreas Fichtner (2021). “Fully Parallel Mesh I/O using PETSc DMPlex with an Application to Waveform Model-

- ing”. In: *SIAM Journal on Scientific Computing* 43.2, pp. C127–C153. DOI: [10.1137/20M1332748](https://doi.org/10.1137/20M1332748). eprint: <http://arxiv.org/abs/2004.08729>.
- Zhang, Junchao, Jed Brown, Satish Balay, Jacob Faibussowitsch, Matthew Knepley, Oana Marin, Richard Tran Mills, Todd Munson, Barry F. Smith, and Stefano Zampini (2022). “The PetscSF Scalable Communication Layer”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4, pp. 842–853. DOI: [10.1109/TPDS.2021.3084070](https://doi.org/10.1109/TPDS.2021.3084070).



# Chapter 5

# Extracting

*The purpose of computing is insight, not numbers.*

— Richard Hamming

## 5.1 Filtering

## 5.2 Submeshes

### `DMPlexCreateSubmesh()`

In order to extract a submesh of lower dimension, we

- Get the join of a set of vertices
- Orient the  $k$ -faces into a Plex
- Find the boundary



## Chapter 6

# Extruding

*In this branch of utopian real estate, architecture is no longer the art of designing buildings so much as the brutal skyward extrusion of whatever site the developer has managed to assemble.*

— Rem Koolhaas

Extrusion is the transformation of a mesh to one of higher dimension replacing each cell by a new cell that has two copies of the original as faces, using a tensor product construction. For example, an extruded segment would become a quadrilateral, and an extruded triangle becomes a triangular prism. Below, we will detail this transformation and the many customizations one can make.

### 6.1 Simple Extrusion

When automating extrusion, we would like to allow for an arbitrary number of layers. Thus we will need dynamic data structures to indicate how cells transform, rather than a static definition. We will imagine that we have a structure with five parts to hold our information,

1. **Nt**: the number of celltypes produced
2. **target**: the celltypes produced
3. **size**: the number of each type of cells
4. **cone**: the cone of each produced cells
5. **ornt**: the orientation of each cone point

The **cone** array is encoded using the scheme we laid out in Section 3.1, namely that each entry has a celltype, the number of cones to be taken, the cone index for each, and finally the replica number. In order to describe extrusion for all cells, we index this structure by celltype. As the simplest example, below we will explain how a vertex is extruded into a series of line segments.

We first indicate that this description is for celltype POINT, and then we set the number of celltypes produced to two. We will make more vertices, and then either SEGMENTS or POINT\_PRISM\_TENSORS, depending on whether we decide to use tensor cells in the extrusion. We use the number of layers, N1, to allocate the rest of our data structure.

---

```

ct = DM_POLYTOPE_POINT;
Nt[ct] = 2;
Nc = 6*N1;
No = 2*N1;
PetscMalloc4(Nt[ct], &target[ct], Nt[ct], &size[ct], Nc, &cone[ct], No, &ornt[ct]);
target[ct][0] = DM_POLYTOPE_POINT;
target[ct][1] = useTensor ? DM_POLYTOPE_POINT_PRISM_TENSOR : DM_POLYTOPE_SEGMENT;
size[ct][0] = N1+1;
size[ct][1] = N1;
for (i = 0; i < N1; ++i) {
    cone[ct][6*i+0] = DM_POLYTOPE_POINT;
    cone[ct][6*i+1] = 0;
    cone[ct][6*i+2] = i;
    cone[ct][6*i+3] = DM_POLYTOPE_POINT;
    cone[ct][6*i+4] = 0;
    cone[ct][6*i+5] = i+1;
}
for (i = 0; i < No; ++i) ornt[ct][i] = 0;

```

---

For N1 layers, we will have N1 segments and N1+1 vertices. The vertices have no cones, but for each segment, we have two vertices which are described solely by replica number since they are all produced by the original vertex. Finally, vertices always have orientation 0.

A less trivial example is provided by the extrusion of triangles. We again produce two celltypes, TRIANGLES and either TRI\_PRISMS or TRI\_PRISM\_TENSORS, and we must specify cones and orientations for both. As before, we make N1 prisms and N1+1 triangles.

---

```

ct = DM_POLYTOPE_TRIANGLE;
Nt[ct] = 2;
Nc = 12*(N1+1) + 18*N1;
No = 3*(N1+1) + 5*N1;
PetscMalloc4(Nt[ct], &target[ct], Nt[ct], &size[ct], Nc, &cone[ct], No, &ornt[ct]);
target[ct][0] = DM_POLYTOPE_TRIANGLE;
target[ct][1] = useTensor ? DM_POLYTOPE_TRI_PRISM_TENSOR : DM_POLYTOPE_TRI_PRISM;
size[ct][0] = N1+1;
size[ct][1] = N1;

```

---

The cones for the triangles are straightforward. They are each formed from the edges produced by the edges of the original triangle, and in the same order. Thus, the replica numbers are exactly the layer number, and the orientations are



zero. If the orientation of the original triangle is nonzero, this will be propagated by the group action mentioned above.

---

```

for (i = 0; i < N1+1; ++i) {
  cone[ct][12*i+0] = DM_POLYTOPE_SEGMENT;
  cone[ct][12*i+1] = 1;
  cone[ct][12*i+2] = 0;
  cone[ct][12*i+3] = i;
  cone[ct][12*i+4] = DM_POLYTOPE_SEGMENT;
  cone[ct][12*i+5] = 1;
  cone[ct][12*i+6] = 1;
  cone[ct][12*i+7] = i;
  cone[ct][12*i+8] = DM_POLYTOPE_SEGMENT;
  cone[ct][12*i+9] = 1;
  cone[ct][12*i+10] = 2;
  cone[ct][12*i+11] = i;
}
for (i = 0; i < 3*(N1+1); ++i) ornt[ct][i] = 0;

```

---

Finally, we construct the cones of the triangular prisms. Each consists of two triangle endcaps, and three side faces which can be either `SEG_PRISM_TENSORS` or `QUADRILATERALS`. The side faces are those extruded by the edges of the original triangle, and are all properly oriented. However, if we make `TRI_PRISMS` then the bottom endcap must reverse orientation so that it has an outward normal.

---

```

coeff = 12*(N1+1);
ooff = 3*(N1+1);
for (i = 0; i < N1; ++i) {
  if (useTensor) {
    cone[ct][coeff+18*i+0] = DM_POLYTOPE_TRIANGLE;
    cone[ct][coeff+18*i+1] = 0;
    cone[ct][coeff+18*i+2] = i;
    cone[ct][coeff+18*i+3] = DM_POLYTOPE_TRIANGLE;
    cone[ct][coeff+18*i+4] = 0;
    cone[ct][coeff+18*i+5] = i+1;
    cone[ct][coeff+18*i+6] = DM_POLYTOPE_SEG_PRISM_TENSOR;
    cone[ct][coeff+18*i+7] = 1;
    cone[ct][coeff+18*i+8] = 0;
    cone[ct][coeff+18*i+9] = i;
    cone[ct][coeff+18*i+10] = DM_POLYTOPE_SEG_PRISM_TENSOR;
    cone[ct][coeff+18*i+11] = 1;
    cone[ct][coeff+18*i+12] = 1;
    cone[ct][coeff+18*i+13] = i;
    cone[ct][coeff+18*i+14] = DM_POLYTOPE_SEG_PRISM_TENSOR;
    cone[ct][coeff+18*i+15] = 1;
    cone[ct][coeff+18*i+16] = 2;
  }
}

```

```

cone[ct][coff+18*i+17] = i;
ornt[ct][ooff+5*i+0] = 0;
ornt[ct][ooff+5*i+1] = 0;
ornt[ct][ooff+5*i+2] = 0;
ornt[ct][ooff+5*i+3] = 0;
ornt[ct][ooff+5*i+4] = 0;
} else {
cone[ct][coff+18*i+0] = DM_POLYTOPE_TRIANGLE;
cone[ct][coff+18*i+1] = 0;
cone[ct][coff+18*i+2] = i;
cone[ct][coff+18*i+3] = DM_POLYTOPE_TRIANGLE;
cone[ct][coff+18*i+4] = 0;
cone[ct][coff+18*i+5] = i+1;
cone[ct][coff+18*i+6] = DM_POLYTOPE_QUADRILATERAL;
cone[ct][coff+18*i+7] = 1;
cone[ct][coff+18*i+8] = 0;
cone[ct][coff+18*i+9] = i;
cone[ct][coff+18*i+10] = DM_POLYTOPE_QUADRILATERAL;
cone[ct][coff+18*i+11] = 1;
cone[ct][coff+18*i+12] = 1;
cone[ct][coff+18*i+13] = i;
cone[ct][coff+18*i+14] = DM_POLYTOPE_QUADRILATERAL;
cone[ct][coff+18*i+15] = 1;
cone[ct][coff+18*i+16] = 2;
cone[ct][coff+18*i+17] = i;
ornt[ct][ooff+5*i+0] = -2;
ornt[ct][ooff+5*i+1] = 0;
ornt[ct][ooff+5*i+2] = 0;
ornt[ct][ooff+5*i+3] = 0;
ornt[ct][ooff+5*i+4] = 0;
}
}

```

---

### 6.1.1 Coordinates

New coordinates for the extruded mesh are determined by following the local normal out some distance from the original surface, and thus can be broken into two parts: computation of the local normal and computation of the layer start and thickness. We shall first look at determining the local normal direction. In the simplest case, we can prescribe a global normal direction, either using the API in `DMPlexExtrude()` and `DMPlexTransformExtrudeSetNormal()`, or using a command line option, `-dm_plex_transform_extrude_normal`. If the normal is not specified and we are extruding an embedded surface, meaning the coordinates come from a higher dimensional space, then we can compute a local normal. The local normal at a mesh point  $p$  is defined to be the average of the cell

normals for all cells contained in its star. The computation of surface normals is described in Section 2.5. It would be possible to weight this average, for example by the area of each cell, but this is not currently done. Finally, if the normal is not computed or specified, the default normal for a two-dimensional extruded mesh is  $\hat{y}$ , and for a three-dimensional mesh is  $\hat{z}$ .

## 6.2 Embedded Extrusion

In crustal dynamics, a *fault* is a crack in the Earth's crust along which there is movement, with adjacent rocks sliding past one another. Faults can be modeled by dislocations in a mesh, cuts allowing faces to slide past each other. In order to model this, we must disconnect the mesh along this plane so that the sides can move independently. We do this by duplicating the  $k$ -cells (faces) which make up the embedded surface. In addition, we can introduce extruded cells to bridge the cut, which forms the basis of the *cohesive cell* method to implement fault rheologies (Aagaard, Knepley, and Williams 2013).

To begin, we must create a representation of the embedded surface and its boundary, which has been described in Chapter 5. We do this using `DMPlexCreateSubmesh()` which accepts a label marking the interface to be split. This is the label passed to `DMPlexCreateHybridMesh()` which calls `DMPlexCreateSubmesh()` internally. On output, four structures describing the new mesh are returned. First, a label indicates what parts of the original mesh impinged on the division surface. Points directly on the surface are labeled with their dimension, so an edge 7 on the division surface would have value 1 in label. Points that impinge from the positive side are labeled with their dimension shifted by one hundred, so an edge 6 with one vertex 3 on the surface would have value 101 and vertex would have value 0. Points from the negative side have the opposite values, so an edge 9 from the negative side of the surface would have value -101 in the label. Second, another label indicates which points in the new mesh were the result of splitting points in the original mesh. The label value is the dimension shifted by  $\pm 100$  depending on the side for the given point. For example, if two edges 10 and 14 in the new mesh result from splitting an edge in the original mesh, the label would have value 101 for edge 10 and value -101 for edge 14. Third, an interface `DM` is built from the original division surface using `DMPlexCreateSubmesh()`. Submeshes all have a label, retrieved using `DMPlexGetSubpointMap()`, which maps each point back to a point in the original mesh from which it was extracted, here the division surface. Lastly, it returns a new mesh with faces on the division surface extruded into prisms, a process we will now detail.

When creating the new mesh, we split the algorithm into two phases: a labeling phase where all decisions are made about how to process points, and a construction phase where the new mesh itself is built based upon those decisions. This is very similar to the strategy used in adaptive refinement in Chapter ???. This division means that in parallel communication may be necessary to build the labels, but the mesh will not change during this time. When we construct

the new mesh, hopefully no communication is needed since the labels contain enough information to proceed independently on each process.

### 6.2.1 Labeling

In order to split the mesh along the embedded surface, we will augment the label constructed in `DMPlexCreateSubmesh()`. We need to mark all cells which are adjacent to the surface so that we can replace points in their cones and supports with duplicates as the surface is pulled apart. We label them exactly as explained above, with negative numbers for one side of the fault, and using a shift added to the point dimension.

We loop over the mesh cells which are in the support of each surface face. The orientation of the surface face in the cell gives the side of the surface, meaning that faces with negative orientation are on the positive side since we always orient for outward normals. For each cell, we iterate over the closure of all its faces. If any point in the closure is on the fault, and is not labeled as being on the surface boundary, we label the face with the same side as the cell. We mark all boundary points as non-replicated, first checking that it is indeed part of the surface label. At this point, we also check for *cross edges*. These are edges in the surface that connect two points on the surface boundary without lying on the boundary themselves. We force these into the boundary as well so that we never have a situation in which an edge is replicated but both of its endpoints are not.

Finally, we must mark any point adjacent to a replicated point on the surface. All these points must lie in the star of surface vertices, and in fact vertices not on the boundary, since the boundary is not replicated. However, we do not have a simple test for determining which side of the surface a given point is on, so we proceed iteratively. For each cell in the star of a surface vertex, we check its faces. Since we have already marked the closure of cells with faces on the surface, cells adjacent to these will have marked faces. This allows us to classify the cell and mark its closure. We iterate this process until the whole vertex star is labeled, or nothing has changed in which case we fail. This iteration cannot fail if each side of the fault is covered by a solid volume of cells. This can be interpreted as a breadth-first search along face connections.

**TODO** If a face is newly marked and shared (owned or ghost), send mark to sharing procs. Use algorithm from `knepley/plex-parallel-submesh`.

### 6.2.2 Construction

Using the label we construct above, we create a new mesh by extruding cells out of our internal surface. The new mesh will be composed of three types of points:

<b>Normal</b>	Points which stay the same
<b>Replicated</b>	Points which are copied
<b>Non-replicated</b>	Points which are not copied, but still make hybrid points
<b>Hybrid</b>	Entirely new points, such as cohesive cells

All points off the surface are normal points. Most surface points are replicated,

except for some boundary points which may be non-replicated. This means that we put the same point on both sides of the split surface, but allow a hybrid cell to bridge them. The hybrid points are tensor cells which connected the replicated and non-replicated points on the surface.

We first calculate the sizes for each depth of each new point type, so that we can determine a numbering. The replicated and hybrid points are placed after the normal points in the numbering. We then set the cone size, support size, and cell type for each new point. The cone sizes and cell types are straightforward, but the support size requires some logic. We use the input label marking all points incident to the fault to determine which side of the surface each connected point is on. This allows us to partition the support of replicated points based on which side of the surface they lie on. After this, we allocate storage in the new DM with `DMSetUp()` and fill in the cones and supports. Then we replace replicated points in the negative side cones. This could have been done as part of the last step, but is simpler to split out. After recreating the coordinates, labels, and point SF, we have a complete mesh.

## References

- Aagaard, Brad T., Matthew G. Knepley, and Charles A. Williams (2013). “A Domain Decomposition Approach to Implementing Fault Slip in Finite-Element Models of Quasi-static and Dynamic Crustal Deformation”. In: *Journal of Geophysical Research: Solid Earth* 118.6, pp. 3059–3079. ISSN: 2169-9356. DOI: [10.1002/jgrb.50217](https://doi.org/10.1002/jgrb.50217).



# Chapter 7

## Refining

*Though the mills of God grind slowly; Yet they grind exceeding small;  
Though with patience He stands waiting, With exactness grinds He all.*

— Henry Wadsworth Longfellow

### 7.1 Regular refinement

We understand the process of refinement as a mesh transformation in which the initial points are replaced by new points with specified cones, exactly as described in Chapter 3. In regular refinement, we replace each point by some number of smaller copies of itself. In the simplest case, each source vertex generates an identical target vertex. A slightly more complicated example is the segment, which is refined to two segments by placing a vertex in the middle,



The endpoints produce identical points, and the segment itself produces the two subsegments and center vertex.

Describe process of figuring out subcell mapping (be precise about orientation mapping):

- Start with identity for orientation 0 (`ex11 -ornt_bounds 0,1 REPLACE=1`)
- Construct another orientation of the cell
- For each cell produced (`tct, r, o`)
  - Loop over cells of type `tct` produced from the original orientation
  - Match them with `DMPolytopeMatchOrientation()`
  - If a match is found, report (`rnew, onew`)
  - `ex11 -print_table 1 -ornt_bounds o,o+1`

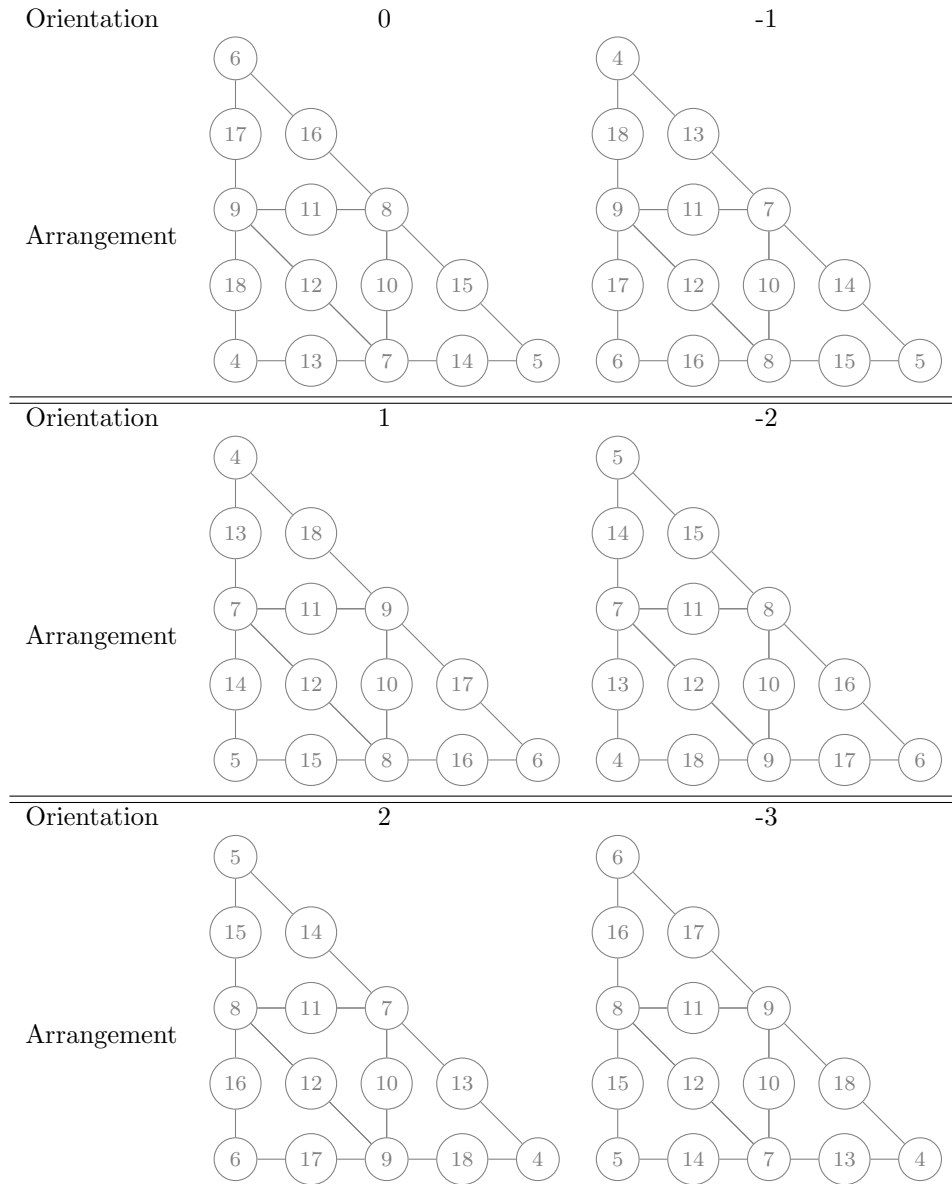


Table 7.1: Regular refinement of all orientations of a triangle.



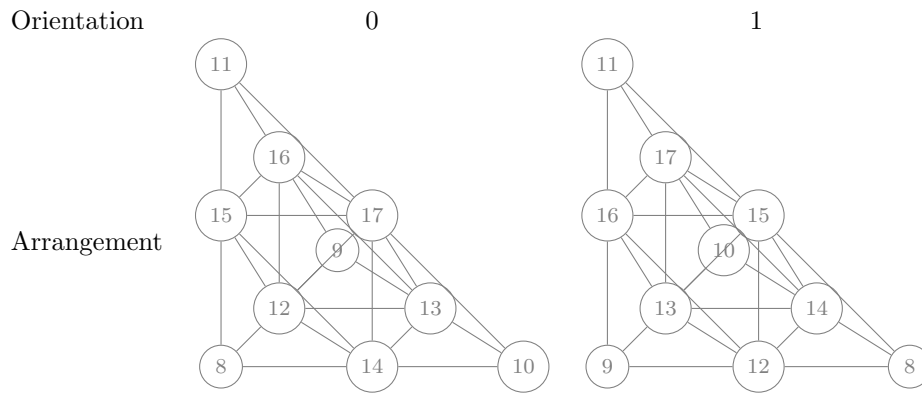
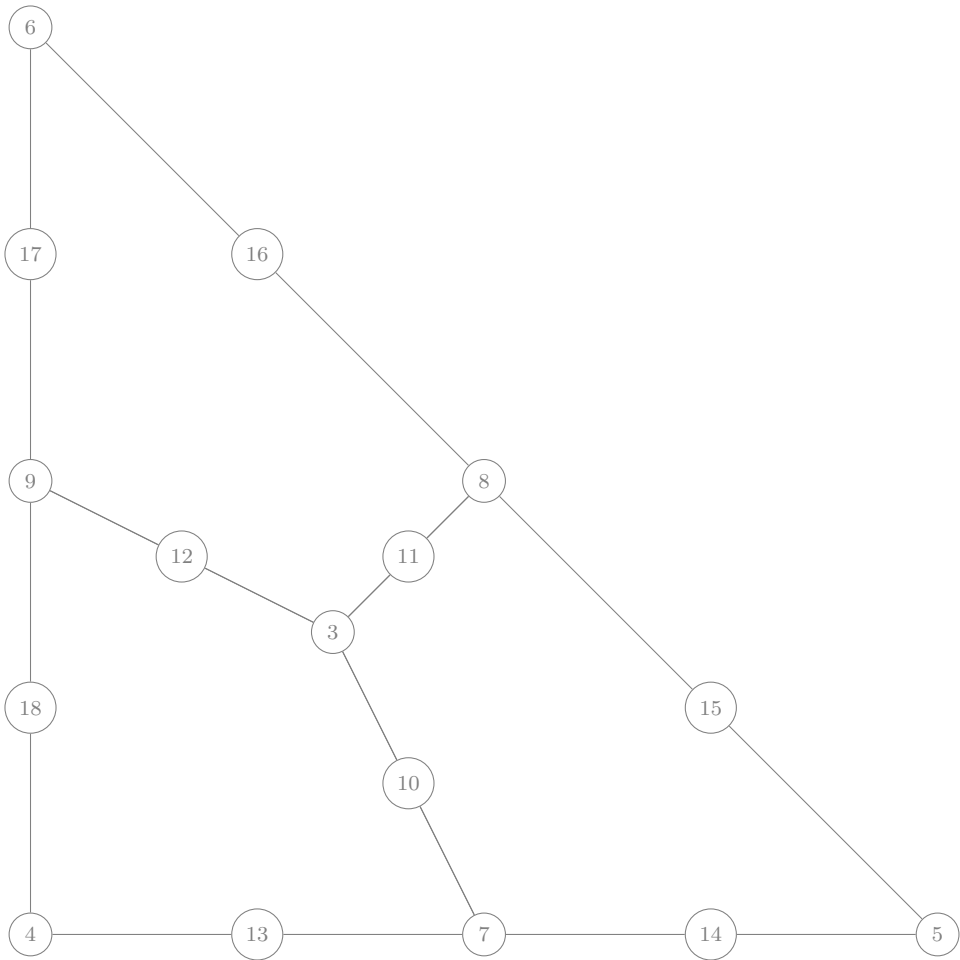
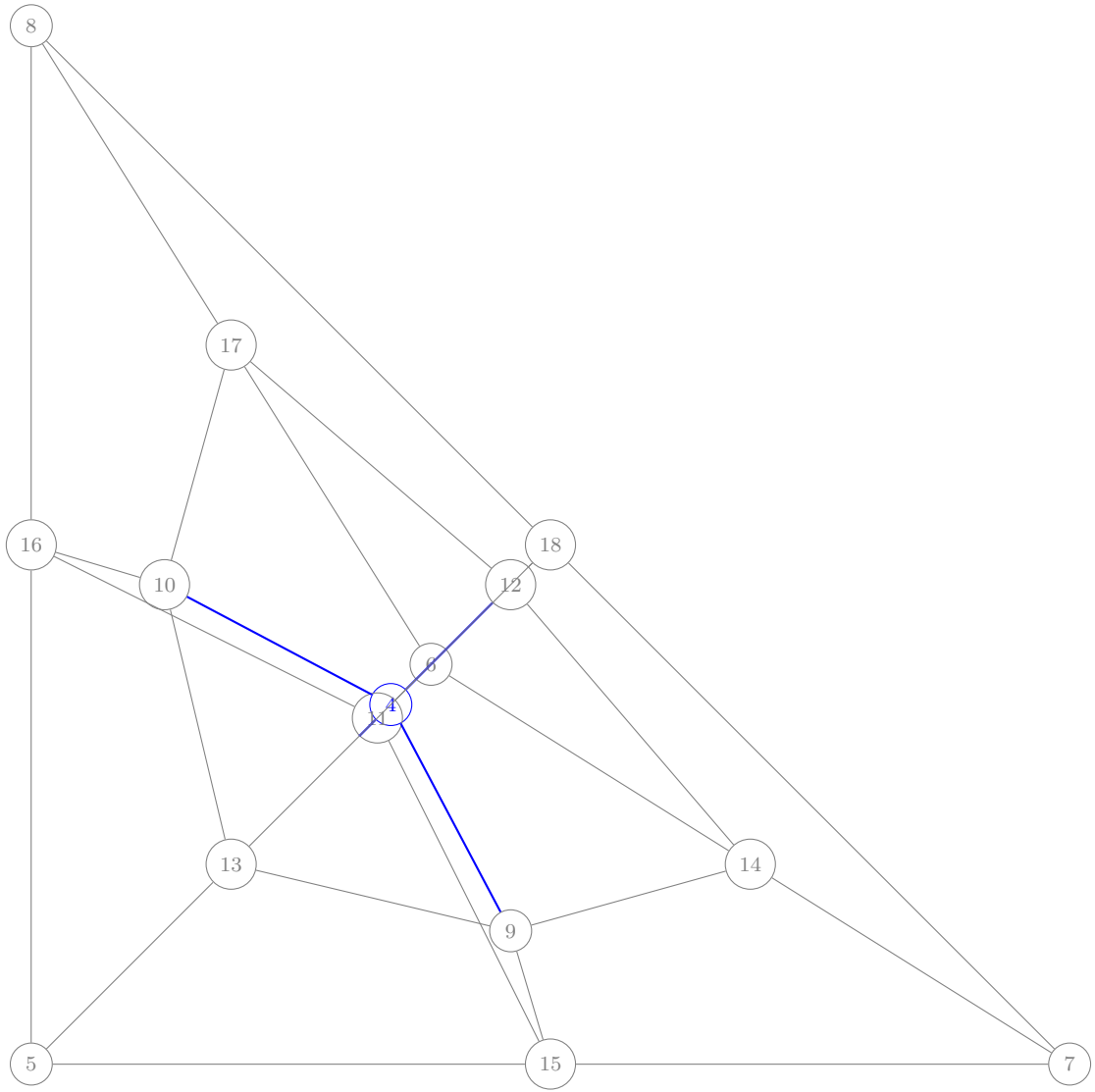


Table 7.2: Regular refinement of all orientations of a tetrahedron.

### 7.1.1 Converting cell types



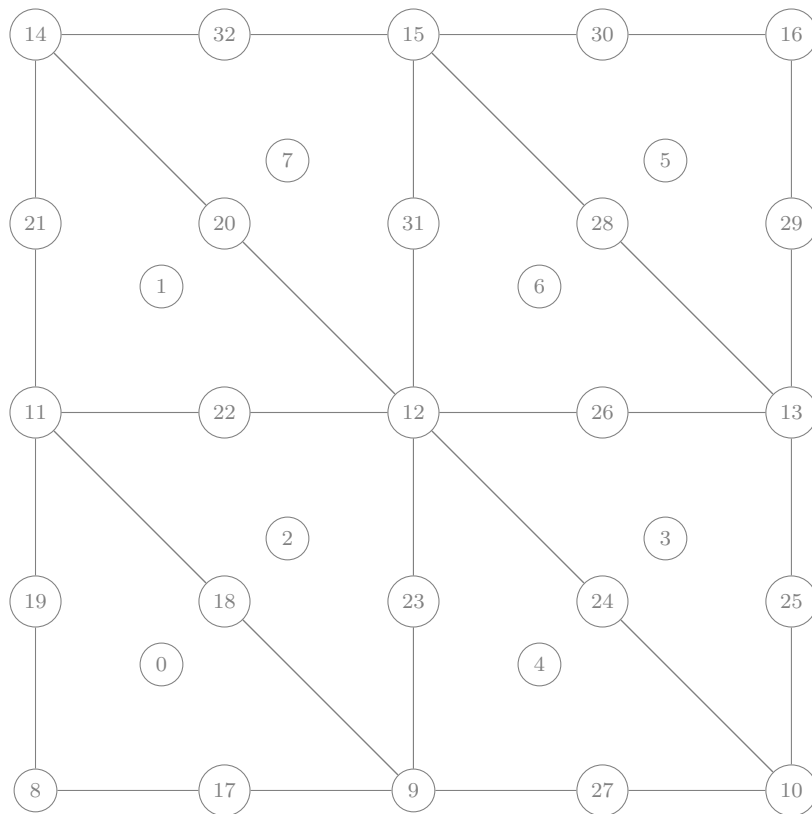


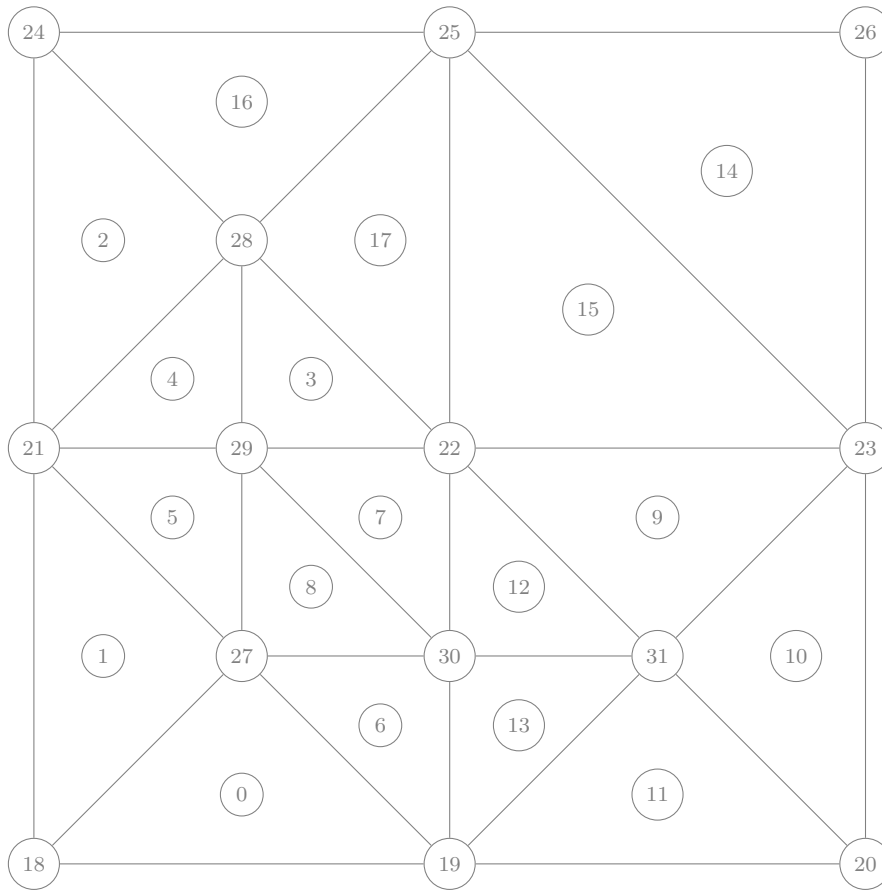
### 7.1.2 Boundary Layers

### 7.1.3 Snapping to structure

## 7.2 Adaptive refinement

### 7.2.1 Plaza



**7.2.2 p4est****7.2.3 ParMMG**



# Chapter 8

# Coordinate Transformations

*The purpose of computing is insight, not numbers.*

— Richard Hamming

## 8.1 Coordinate Representation

## 8.2 Direct Modification

Coordinates in Plex are directly available to the user. If the FE discretizing coordinates is interpolatory, simple modifications can be accomplished by directly modifying the coordinate coefficient values. For example, one could shift a mesh along a given vector  $\mathbf{v}$  using

---

```
Vec x1;
PetscScalar *coords;
PetscInt cdim, N;

DMGetCoordinateDim(dm, &cdim);
DMGetCoordinatesLocal(dm, &x1);
VecGetLocalSize(x1, &N);
VecGetArrayWrite(x1, &coords);
for (PetscInt p = 0; p < N/cdim; ++p) {
    for (PetscInt d = 0; d < cdim; ++d) coords[p * cdim + d] += v[d];
}
VecRestoreArrayWrite(x1, &coords);
DMSetCoordinates(dm, NULL);
```

---

The last line destroys the global coordinate vector, so that it will be recreated on the next call to `DMGetCoordinates()` using the new values from the local vector. We could have instead changed the global coordinate values and recreated the local vector.

### 8.3 Projection

More complex coordinate transformations, or coordinate update for complex coordinate spaces can use a different mechanism, which amounts to a projection into the coordinate space. The function which accomplishes the transformation is `DMPlexRemapGeometry()`, which takes a `PetscPointFunc` argument describing the map. As a simple example, we will shear a domain in direction  $\mathbf{w}$ , meaning

$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{m}(\mathbf{x} \cdot \mathbf{w}). \quad (8.1)$$

where the vector  $\mathbf{m}$  indicates the strength of the shearing in each direction, and should be zero in the direction of  $\mathbf{w}$ . For example, to shear in the  $z$  direction, we would apply

$$\begin{pmatrix} 1 & 0 & m_x \\ 0 & 1 & m_y \\ 0 & 0 & 1 \end{pmatrix} \quad (8.2)$$

to the coordinate vector  $\mathbf{x}$ .

First we will create our strength, or moduli, vector  $\mathbf{m}$  from an array of multipliers, assuming that the direction  $\mathbf{w}$  is one of the coordinate directions `dir`, which is added at the beginning as the first modulus,

---

```

DM cdm;
PetscDS cds;
PetscInt cdim;
PetscScalar *moduli;

DMGetCoordinateDM(dm, &cdm);
DMGetCoordinateDim(dm, &cdim);
PetscMalloc1(cdim + 1, &moduli);
moduli[0] = dir;
for (PetscInt d = 0, e = 0; d < cdim; ++d)
    moduli[d + 1] = d == dir ? 0.0 : multipliers[e++];
DMGetDS(cdm, &cds);
PetscDSSetConstants(cds, cdim + 1, moduli);

```

---

After creating the moduli, we set them as constants in the `PetscDS` associated with the coordinate DM. That way, these will be passed into our point function during the projection operation. Now we need to define a point function for the transformation

---

```

void f0_shear(PetscInt dim, PetscInt Nf, PetscInt NfAux,
             const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u[],
             const PetscScalar u_t[], const PetscScalar u_x[],
             const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a[],
             const PetscScalar a_t[], const PetscScalar a_x[],

```



```
PetscReal t, const PetscReal x[], PetscInt numConstants,  
const PetscScalar constants[], PetscScalar coords[])  
{  
    const PetscInt Nc = uOff[1] - uOff[0];  
    const PetscInt ax = (PetscInt)PetscRealPart(constants[0]);  
    PetscInt c;  
  
    for (c = 0; c < Nc; ++c) coords[c] = u[c] + constants[c + 1] * u[ax];  
}
```

---

After this setup, we can remap the coordinates and free the moduli vector.

---

```
DMPlexRemapGeometry(dm, 0.0, f0_shear);  
PetscFree(moduli);
```

---

In the library, this transformation has been encapsulated in the function `DMPlexShearGeometry()`.



**Part III**

**Applications**



## Chapter 9

# Crustal Dynamics

*The purpose of computing is insight, not numbers.*

— Richard Hamming



## Chapter 10

# Low Mach Flow

*The purpose of computing is insight, not numbers.*

— Richard Hamming





# Appendices



# Appendix A

## Creating DMPlex Meshes

The simplest way to create a DMPlex is shown below:

---

```
DMCreate(comm, &dm);
DMSetType(dm, DMPLEX);
DMSetFromOptions(dm);
DMViewFromOptions(dm, NULL, "--dm-view");
```

---

This first creates an empty DM object, and then sets its implementation type to “plex”. Remember that PETSc objects have a two layer structure, with the top layer being the generic interface (DM), and the bottom layer containing the concrete implementation type (DMPlex) (Balay et al. 2022). From this basis, you can use command line options to construct, manipulate, and output meshes.

### A.1 Working with files

It is very common to use a mesh generator, such as Gmsh (Geuzaine and Remacle 2009), TetGen (Si 2015) or Cubit Blacker, Bohnhoff, and Edwards 1994, or CAD program, or simulation package, and end up with your mesh in a file. PETSc can read a large number of format and convert them to a Plex, listed in Table A.1. You can read in a file in any of these formats using

```
-dm_plex_filename <name>
```

or read in a mesh boundary and have the mesh generated automatically using

```
-dm_plex_boundary_filename <name>
```

Once the file is read in, the Plex can become the object of any of the available transformations. For example, Gmsh meshes are stored with only cells and vertices,

```
$ cd src/dm/impls/plex/tests
$ make ex1
$ ./ex1 -dm_plex_filename ${PETSC_DIR}/share/petsc/datafiles/meshes/square.msh -dm_view
-dm_plex_interpolate 0
DM Object: Simplicial Mesh 1 MPI processes
```

```

type: plex
Simplicial Mesh in 2 dimensions:
  0-cells: 30
  2-cells: 42
Labels:
  celltype: 2 strata with value/size (3 (42), 0 (30))
  depth: 2 strata with value/size (0 (30), 1 (42))
  Cell Sets: 1 strata with value/size (7 (42))

```

Notice that there are 30 vertices (0-cells) and 42 triangles (2-cells with celltype 3). We can read one in and have the edges and faces calculated automatically using the interpolation transformation,

```

$ ./ex1 -dm_plex_filename ${PETSC_DIR}/share/petsc/datafiles/meshes/square.msh -dm_view
-dm_plex_interpolate 1
DM Object: Simplicial Mesh 1 MPI processes
type: plex
Simplicial Mesh in 2 dimensions:
  0-cells: 30
  1-cells: 71
  2-cells: 42
Labels:
  celltype: 3 strata with value/size (0 (30), 3 (42), 1 (71))
  depth: 3 strata with value/size (0 (30), 1 (71), 2 (42))
  Cell Sets: 1 strata with value/size (7 (42))
  Face Sets: 4 strata with value/size (11 (4), 8 (4), 10 (4), 9 (4))

```

so now there are also 71 edges (1-cells). We can read meshes in parallel, and by default this happens on the root process,

```

$ mpiexec -np 3 ./ex1 -dm_plex_filename ${PETSC_DIR}/share/petsc/datafiles/meshes/square.msh
-dm_view
DM Object: Simplicial Mesh 3 MPI processes
type: plex
Simplicial Mesh in 2 dimensions:
  0-cells: 30 0 0
  1-cells: 71 0 0
  2-cells: 42 0 0
Labels:
  celltype: 3 strata with value/size (0 (30), 3 (42), 1 (71))
  depth: 3 strata with value/size (0 (30), 1 (71), 2 (42))
  Cell Sets: 1 strata with value/size (7 (42))
  Face Sets: 4 strata with value/size (11 (4), 8 (4), 10 (4), 9 (4))

```

but we can tell PETSc to distribute the mesh over processes, using a certain partitioner,

```

$ mpiexec -np 3 ./ex1 -dm_plex_filename ${PETSC_DIR}/share/petsc/datafiles/meshes/square.msh
-dm_view -dm_distribute -petscpartitioner_type parmetis
DM Object: Simplicial Mesh 3 MPI processes
type: plex
Simplicial Mesh in 2 dimensions:
  0-cells: 13 13 13
  1-cells: 26 26 26
  2-cells: 14 14 14
Labels:
  depth: 3 strata with value/size (0 (13), 1 (26), 2 (14))
  celltype: 3 strata with value/size (0 (13), 1 (26), 3 (14))
  Cell Sets: 1 strata with value/size (7 (14))
  Face Sets: 2 strata with value/size (8 (2), 10 (4))

```

which gives us 14 triangles on each process. We can visualize this most flexibly using HDF5 and Paraview

```

$ mpiexec -np 3 ./ex1 -dm_plex_filename ${PETSC_DIR}/share/petsc/datafiles/meshes/square.msh
-dm_distribute -petscpartitioner_type parmetis -dm_partition_view -dm_view hdf5:mesh.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh.h5

```

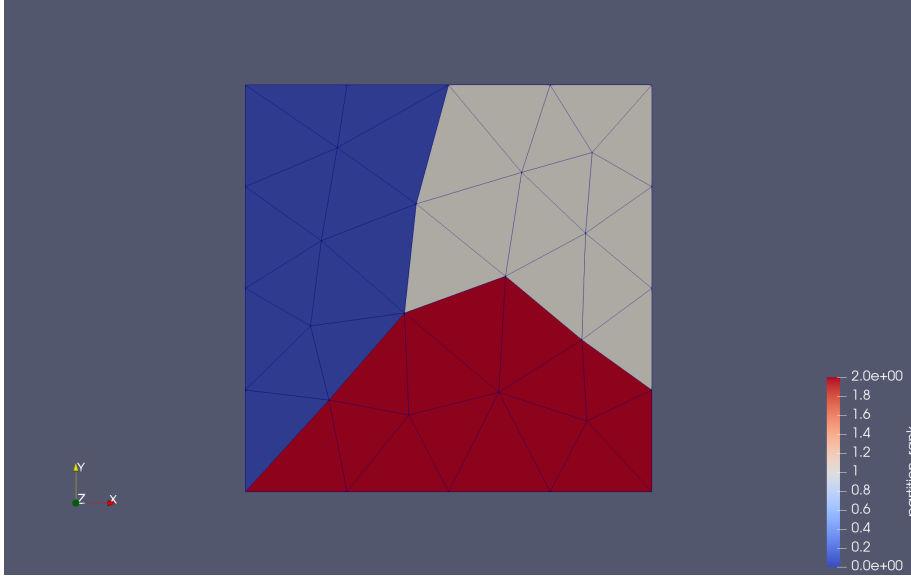


Figure A.1: Gmsh mesh partitioned and distributed by PETSc.

which produces Fig. A.1. Next, we will look at creating meshes from familiar shapes.

## A.2 Working with Shapes

Plex has a few built-in domain shapes for which it can create meshes automatically,

DM_SHAPE_BOX	The tensor product of intervals in dimension $d$
DM_SHAPE_BOX_SURFACE	The surface of a box in dimension $d + 1$
DM_SHAPE_BALL	The $d$ -dimensional ball
DM_SHAPE_SPHERE	The surface of the $(d + 1)$ -dimensional ball
DM_SHAPE_CYLINDER	The tensor product of the interval and disk

which user can select using `-dm_plex_shape`. They can also choose to create a domain with the shape of any of the reference cells by using `-dm_plex_reference_cell_domain`. Even after choosing the shape, there remain many properties to be specified for the mesh. By default, meshes are two-dimensional, but this can be changed using `-dm_plex_dim`. Meshes are also interpolated by default, but this can be set using `-dm_plex_interpolate`. The coordinates can be scaled with `-dm_plex_scale`. Last, we need to specify what kind of cells will discretize the domain. The most common choices are either simplices or tensor product cells, which can be set using `-dm_plex_simplex`. However, some shapes support more exotic cells, notably the reference cell domain, which can be set directly using `-dm_plex_cell`.

Format	Extension
PETSc/HDF5	.h5
PETSc/Cell-Vertex	.dat
Gmsh	.msh
Gmsh2	.msh2
Gmsh4	.msh4
CGNS	.cgns
ExodusII	.exo
Genesis	.gen
Fluent	.cas
Med	.med
PLY	.ply
STEP	.stp
IGES	.iges
EGADS	.egads
EGADSLite	.egadslite

Table A.1: File formats that can be read and converted to a DMPlex.

**Example: Box** The simplest, but probably most useful built-in shape is the box. We can control the number of faces in each direction, and the bounding box corners through options. The run below produces the mesh in Fig. A.2.

```
$ ./ex1 -dm_plex_shape box -dm_plex_dim 3 -dm_plex_box_faces 1,3,5 -dm_plex_box_lower -1,-1,-1
-dm_plex_box_upper 1,1,1 -dm_view hdf5:mesh.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh.h5
```

**Example: Periodic box** We can also control the periodicity of each dimension. The default behavior in Plex is to represent periodicity by making a periodic topology, rather than identifying separate mesh points. Here we make a square mesh, periodic in  $x$  and  $y$  using tensor cells, and demonstrate two different embeddings, as shown in Fig. A.3.

```
$ ./ex1 -dm_plex_shape box -dm_plex_simplex 0 -dm_plex_box_faces 50,70
-dm_plex_box_bd periodic,periodic -dm_view hdf5:mesh.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh.h5
```

We can also cut along the periodic boundary to unroll the mesh for viewing. Below we show a mesh periodic only in  $x$ , with and without the cut, in Fig. A.4

```
$ ./ex1 -dm_plex_shape box -dm_plex_simplex 0 -dm_plex_box_faces 5,7
-dm_plex_box_bd periodic,none -dm_view hdf5:mesh1.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh1.h5
$ ./ex1 -dm_plex_shape box -dm_plex_simplex 0 -dm_plex_box_faces 5,7
-dm_plex_box_bd periodic,none -dm_plex_periodic_cut -dm_view hdf5:mesh2.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh2.h5
```

**Example: Box with wedges** For some shapes, we can change the type of cell we use. Above, we showed both simples and tensor cells for boxes. We can also prisms, as shown in Fig. A.5.

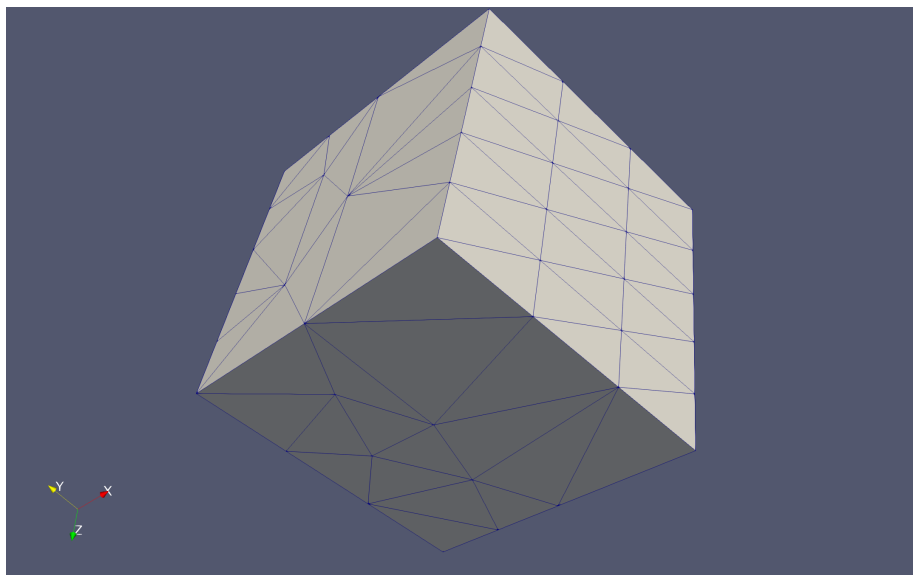


Figure A.2: Box mesh with tetrahedra.

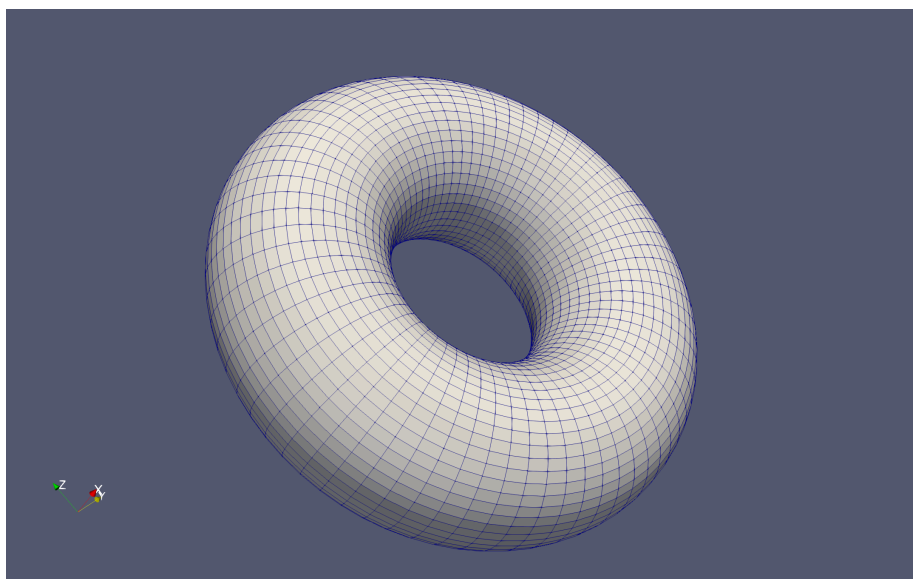


Figure A.3: Doubly periodic box mesh with quadrilaterals.

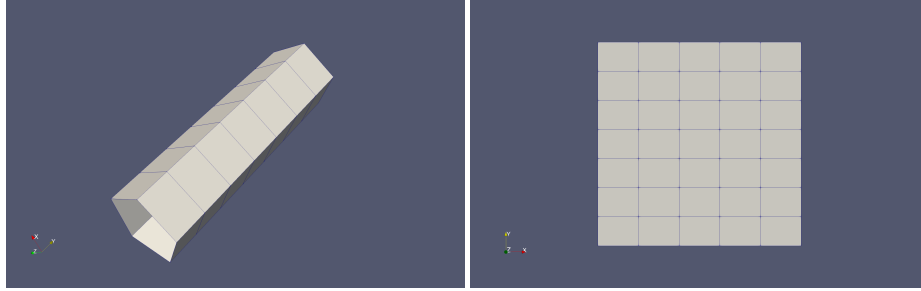


Figure A.4: Singly periodic box mesh with quadrilaterals, embedded (left) and cut (right).

```
$ ./ex1 -dm_plex_shape box -dm_plex_dim 3 -dm_plex_cell tensor_triangular_prism
-dm_plex_box_faces 5,7,3 -dm_view hdf5:mesh.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh.h5
```

**Example: Sphere** Plex creates a sphere using the icosahedral mesh as a starting point. If we refine it, the subsequent points are inserted on the surface, as shown in Fig. A.6.

```
$ ./ex1 -dm_plex_shape sphere -dm_plex_sphere_radius 10 -dm_view hdf5:mesh1.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh1.h5
$ ./ex1 -dm_plex_shape sphere -dm_plex_sphere_radius 10 -dm_refine 2 -dm_view hdf5:mesh2.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh2.h5
```

We can also mesh spheres using tensor cells, where we begin with a cube, as in Fig. A.7.

```
$ ./ex1 -dm_plex_shape sphere -dm_plex_simplex 0 -dm_view hdf5:mesh1.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh1.h5
$ ./ex1 -dm_plex_shape sphere -dm_plex_simplex 0 -dm_refine 4 -dm_view hdf5:mesh2.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh2.h5
```

**Example: Ball** The ball can be created by running a mesh generator on the sphere mesh, as in Fig. A.8. The generated mesh would be very poor if we started with a coarse boundary, so we refine it twice using the `bd_` prefix. Notice that by default, the generator does not create a nice interior mesh, and it will look worse with uniform refinement, so we refine it based on a volume constraint up front.

```
$ ./ex1 -dm_plex_shape ball -dm_plex_dim 3 -dm_plex_ball_radius 10 -bd_dm_refine 2
-dm_view hdf5:mesh1.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh1.h5
$ ./ex1 -dm_plex_shape ball -dm_plex_dim 3 -dm_plex_ball_radius 10 -bd_dm_refine 2
-dm_refine_volume_limit_pre 10 -dm_view hdf5:mesh2.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh2.h5
```

It would be possible to use tensor cells in a cubed-sphere approach, but PETSc does not currently support this.



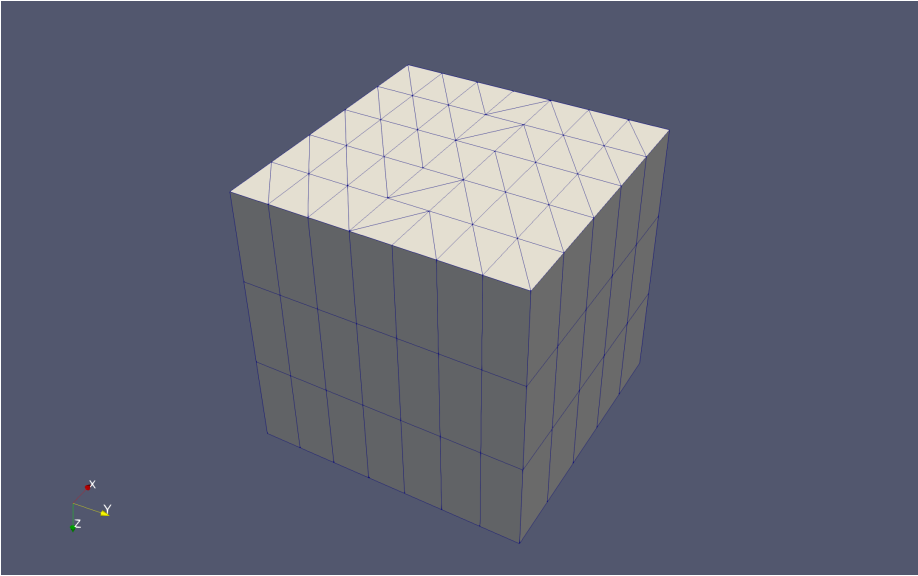


Figure A.5: Box mesh with tensor triangular prisms.

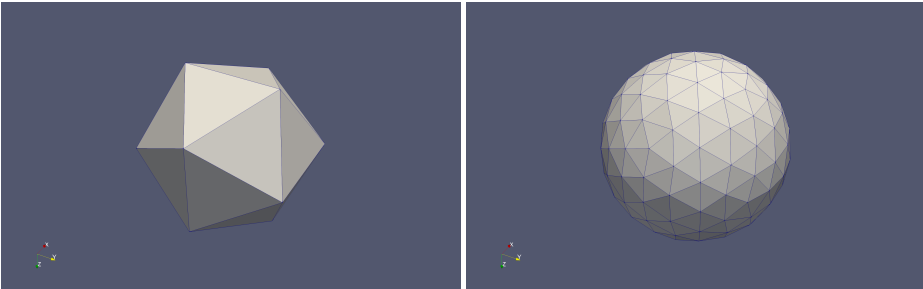


Figure A.6: Sphere mesh with triangles, before and after refinement.

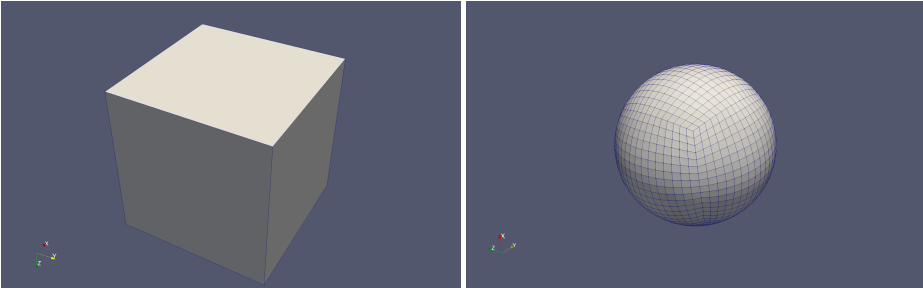


Figure A.7: Sphere mesh with quadrilaterals, before and after refinement.

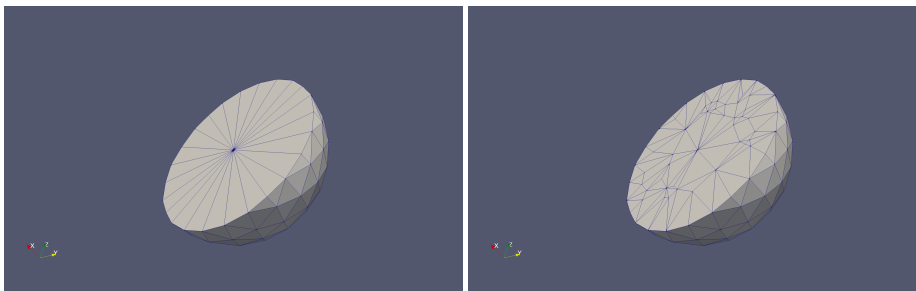


Figure A.8: Ball mesh with tetrahedra, before and after volume-constrained refinement.

**Example: Periodic cylinder** Currently, Plex only creates cylinders with tensor cells so that periodicity is supported, but visualization for periodic cylinders is not yet supported. We can make a non-periodic cylinder, but refinement remapping is only effective once, so we turn it off during a pre-refinement stage, with the final cylinder shown in Fig. A.9.

```
$ ./ex1 -dm_plex_shape cylinder -dm_refine_pre 2 -dm_refine_remap_pre 0 -dm_refine 1
  -dm_view hdf5:mesh.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh.h5
$ ./ex1 -dm_plex_shape cylinder -dm_plex_cylinder_bd periodic
```

**Example: Cylinder with wedges** We can make a cylinder from tensor triangular prisms, as in Fig. A.10. Notice that refinement for tensor cells does not split the tensor direction.

```
$ ./ex1 -dm_plex_shape cylinder -dm_plex_cell tensor_triangular_prism -dm_plex_cylinder_num_wedges 7
  -dm_refine 1 -dm_view hdf5:mesh.h5
$ ${PETSC_DIR}/lib/petsc/bin/petsc_gen_xdmf.py mesh.h5
```

### A.3 Working in Parallel

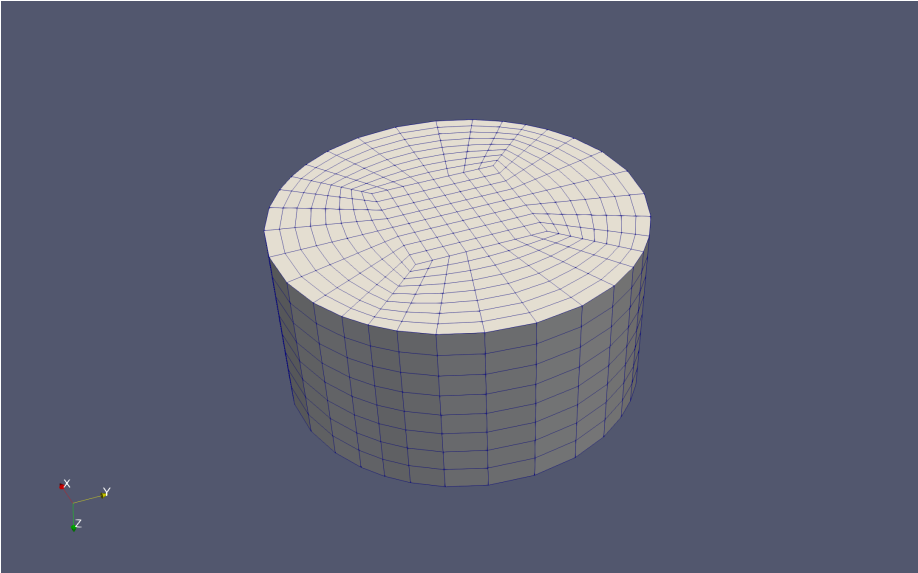


Figure A.9: Cylinder mesh with tensor triangular prisms.

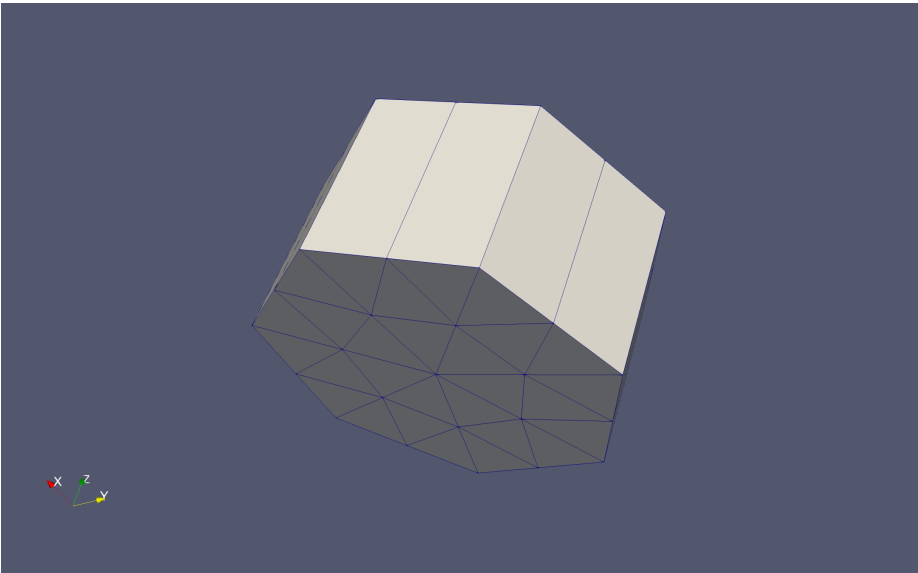


Figure A.10: Cylinder mesh with tensor triangular prisms.

## References

- Balay, Satish et al. (2022). *PETSc/TAO Users Manual*. Tech. rep. ANL-21/39 - Revision 3.18. Argonne National Laboratory.
- Geuzaine, Christophe and Jean-François Remacle (2009). “Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities”. In: *International Journal for Numerical Methods in Engineering* 79.11, pp. 1309–1331.
- Si, Hang (Feb. 2015). “TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator”. In: *ACM Trans. on Mathematical Software* 41.2. DOI: [10.1145/2629697](https://doi.org/10.1145/2629697).
- Blacker, Ted D, William J Bohnhoff, and Tony L Edwards (1994). *CUBIT mesh generation environment. Volume 1: Users manual*. Tech. rep. Sandia National Labs., Albuquerque, NM (United States).

## Index

Numbers written in *italic* refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in *roman* refer to the pages where the entry is used.

<i>k</i> -cell,	<u>11</u>	fault,	<u>67</u>	orientation,	<u>15</u> , <u>16</u>
Cayley Table,	<u>21</u>	group transformations,	<u>15</u>	output sensitive,	<u>45</u>
cohesive cell,	<u>67</u>	hybrid,	<u>55</u>	reduced point number,	<u>51</u>
complex, <u>11</u> ,	<u>16</u>	interpolation,	<u>55</u>	replica number,	<u>48</u>
configuration,	<u>14</u>	invariance theorems,	<u>15</u>	root,	<u>57</u>
conforming,	<u>11</u>	leaf,	<u>57</u>	topological space,	<u>11</u>
Coxeter groups,	<u>21</u>	mesh interpolation,	<u>16</u>	transformation type,	<u>47</u>
cross edges,	<u>68</u>			weakly equivalent,	<u>41</u>
dihedral group,	<u>19</u>				
dof,	<u>39</u>				