SPACETIME MESHING OF STRATIFIED SPACES FOR SPACETIME DISCONTINOUS GALERKIN METHODS IN ARBITRARY SPATIAL DIMENSIONS

BY

CHRISTIAN JOSEPH HOWARD

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Jeff Erickson

ABSTRACT

We introduce the spacetime discontinuous Galerkin method and motivate the need for supporting spacetime meshing on meshes comprised of multiple manifolds. We first discuss preliminary concepts behind simplices, simplicial complexes, and the generalization to oriented simplicies. Using these ideas, we define *stratified spaces* and how they can be used to model a mesh comprised of multiple oriented manifolds. We construct a graphical representation called a *Stratified Mesh* and use this representation to construct a collection of data structures, the main result being the StratifiedMesh data structure. Next we define a set of support algorithms based on the various data structures discussed. This leads us to review the fundamentals of the TentPitcher algorithm and its relationship to spacetime discontinuous Galerkin methods both theoretically and in the literature. The TentPitcher algorithm is then extended to work on stratified meshes in $\mathbb{E}^d \times \mathbb{R}$ for arbitrary spatial dimension d. We then briefly discuss a parametrization for tentpole vertices that generalizes the baseline TentPitcher , vertex smoothing, and tilted tentpoles. Following that, we discuss at a high level the generic software architecture and techniques used build completely new spacetime meshing software that handles stratified meshes. Visualizations of various examples from the software conclude the work, with examples of single manifold $2d \times$ time, single manifold $3d \times$ time, and a multiple manifold example in $2d \times$ time.

To my awesome dog, great friends, and amazing family.

ACKNOWLEDGMENTS

I am thankful for the time and patience that my advisors Jeff Erickson and Bob Haber have had with me as I have worked on this project and I especially thank Jeff for the time he has taken to give me feedback on this thesis. I am also thankful to them both for our fruitful discussions over the last year and a half. These discussions have greatly influenced the work presented in this thesis and have educated me in some generally useful ways. I would certainly like to thank other interactions I have had within the group and especially thank Amit Madhukar for time he has put aside to meet up with me and talk with me when I felt it was needed.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

Within the domain of Scientific Computing, models driven by differential equations are commonplace. For a myriad of problem domains, partial differential equations (PDEs) and their corresponding boundary conditions dominate as the way phenomena of interest can be compactly represented and eventually solved for by engineers and scientists. Some of the most fascinating phenomena that are studied involve wave-like behavior that can be captured by the specific class of hyperbolic PDEs. Such models find use in describing everything from flow and shockwaves of inviscid fluids to sound propagation, electromagnetic phenomenon, and crack propagation in various structures. These models enjoy the intuitive feature that information travels within the domain at finite speeds, just as a leaf resting on the surface of a river might travel down said river at a finite speed. Ironically, this property also naturally leads to the potential for discontinuous solutions to hyperbolic PDEs, making them some of the most challenging problems to solve numerically with high accuracy. Fortunately, the fundamental properties behind hyperbolic PDEs can be exploited by discretizing spacetime directly, resulting in elegant algorithms that take advantage of this property.

The spacetime discontinuous Galerkin (SDG) method is a finite element method that takes advantage of the finite information propagation of hyperbolic PDEs in a way that is not seen in other methods tackling the same problems, such as Finite Difference, Finite Volume, and Discontinuous Galerkin methods that discretize only in space [1, 2]. Given a spatial domain $\Omega \subset \mathbb{E}^d$ and a time interval $[0, t] \subset \mathbb{R}$, this method aims to directly tackle hyperbolic systems of equations in the domain $D = \Omega \times [0, t]$, where the system of Hyperbolic equations are typically of the form [1]

$$\frac{\partial \boldsymbol{v}}{\partial t} + \sum_{i=1}^{d} \frac{\partial}{\partial x_i} \boldsymbol{F}^{(i)}(\boldsymbol{v}) = \boldsymbol{g}(t, \boldsymbol{x}) \tag{1.1}$$

where $\boldsymbol{v} = \boldsymbol{v}(t, \boldsymbol{x}) = [v_1(t, \boldsymbol{x}), \cdots, v_n(t, \boldsymbol{x})]^T$ is our unknown, $\boldsymbol{F}^j(\boldsymbol{v})$ are known flux functions, and $\boldsymbol{g}(t, \boldsymbol{x})$ is known. Spacetime Discontinuous Galerkin, unlike other techniques, discretizes spacetime in order to solve the system of PDEs. There are numerous advantages to this, such as producing solution fields that can be evaluated at any time and algorithms, as will be explained later, that promise very efficient solutions to the PDEs. The challenge with this technique, however, is meshing in spacetime is more complicated than typical meshing approaches due to our need to satisfy causality conditions tied to the characteristics of the hyperbolic system. However, much progress has been made in this endeavor [3, 4, 5, 6, 7], particularly for domains represented as a single manifold. Rigorously generalizing the spacetime meshing algorithms to problems comprised of multiple manifolds has not been given much attention, however. These problems naturally arise in a variety of domains. One application of interest is modeling crack propagation where, for example, the domain is some rectangle and the crack is initially some line within the rectangle that evolves over time using a damage model while the rectangular domain evolves using models from linear elasticity.

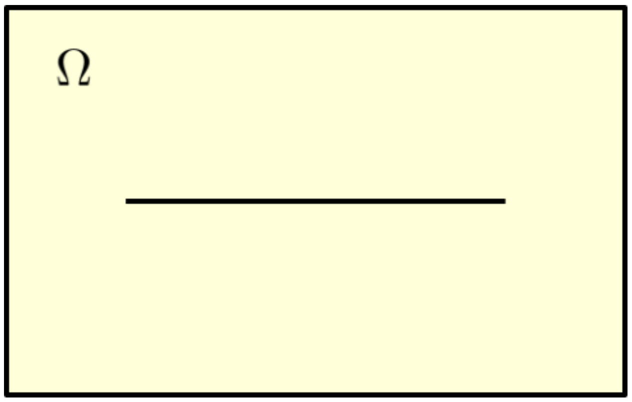

Figure 1.1: Example $2d$ rectangular mesh with a crack in the center

In this thesis, we extend this work to *stratified spaces*, which consist of multiple manifolds of mixed dimension. The extension to stratified spaces enables the SDG method to investigate problems like crack propagation and others that are represented by domains with multiple manifolds of mixed dimensions.

1.2 OVERVIEW

Within this thesis, we will start by reviewing some mathematical preliminaries that we will need before we discuss *stratified spaces* and their role in this work. These preliminary topics are primarily about simplicies, simplicial complexes, and the generalization to oriented simplices using basics from simplicial homology. Following these preliminary topics, we define stratified spaces and discuss how we might model them in the context of meshing, eventually stepping toward a graphical model called the *stratified mesh* that will prove convenient in data structure design.

The following chapter focuses on data structure design for the meshing objects that will be used to model the stratified meshes. We also touch on algorithms tied to these data structures and eventually find our way to the TentPitcher algorithm. Once here, we tie the stratified mesh to the TentPitcher algorithm and move into a discussion about software implementation and some sample results.

CHAPTER 2: SPACETIME MESHING WITH MULTIPLE MANIFOLDS

2.1 PLAYING WITH SIMPLICES, COMPLEXES, AND HOMOLOGY

Let us consider the space $\mathbb{R}^d$ and define $V = \{\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_m\} \subset \mathbb{R}^d$ to be a collection of points represented as column vectors. A *convex combination* of V is a point $q = \sum_{i=0}^{m} \lambda_i \boldsymbol{v}_i$ where $\lambda_i \geq 0$ and $\sum_{i=0}^{m} \lambda_i = 1$. The *convex hull* of V is the set of *all* convex combinations of V, where the resulting set is defined as a *convex polytope* and the elements of V are the *vertices* of that polytope. The dimension of the convex polytope comprised of vertices in V is equivalent to $\dim V = \operatorname{rank}([\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_m])$. A d-polytope is defined as a polytope where $\dim V = d$. A d-simplex Δ is a d-polytope with the special case $|V| = (d+1)$. We represent a d-simplex by its vertex set $V = \Delta = \{\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_d\}$. A degenerate simplex is one where $\dim \Delta < |\Delta| - 1$. In our work, we assume every simplex is not degenerate, meaning that $\dim \Delta = |\Delta| - 1$. A *face* f of a d-simplex Δ is any simplex such that $f \subseteq \Delta$ and a *facet* is any face f with $\dim f = \dim \Delta - 1 = (d-1)$. A *coface* of a k-simplex f is any simplex Δ such that $f \subseteq \Delta$ and a *cofacet* is a coface Δ with $\dim \Delta = \dim f + 1$. We denote the set of faces, facets, cofaces, and cofacets of an oriented simplex Δ to be $\textbf{faces}(\Delta)$, $\textbf{facets}(\Delta)$, $\textbf{cofaces}(\Delta)$, and $\textbf{cofacets}(\Delta)$. The subset of faces of dimension k is represented as $\textbf{faces}_k(\Delta)$. For a more thorough treatment of convex polytopes, simplices, and related topics, readers can refer to standard references [8, 9].

In simplicial homology, we generalize a simplex Δ to an oriented d-simplex $\vec{\Delta}$ by representing it as the sequence $(\boldsymbol{v}_0, \cdots, \boldsymbol{v}_d)$ of length $(d+1)$ [10, 11]. The dimension of $\vec{\Delta}$ is defined in the same way as the non-oriented simplex, namely $\dim \vec{\Delta} = \operatorname{rank}([\boldsymbol{v}_0, \boldsymbol{v}_1, \cdots, \boldsymbol{v}_m])$ for the vertices that form $\vec{\Delta}$. Two oriented simplices $\vec{\Delta}$ and $\vec{\Delta}'$ are equivalent if $\vec{\Delta}'$ is an *even*

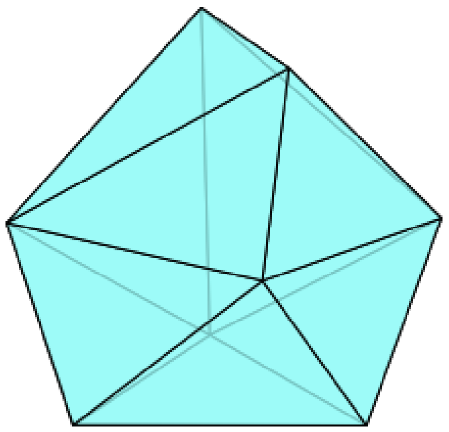
Figure 2.1: Example convex 3-polytope

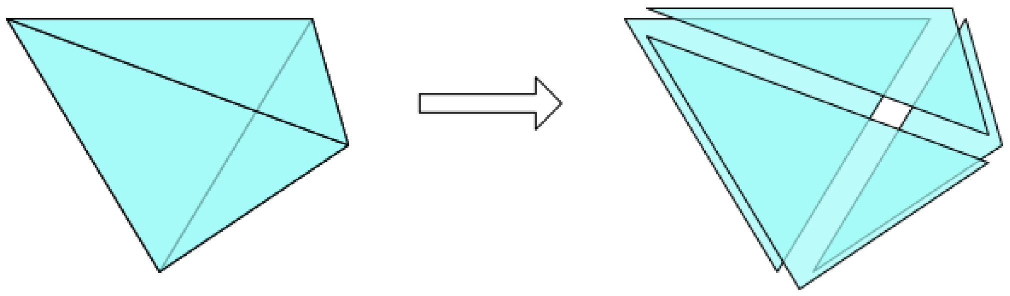
Figure 2.2: Example 3-simplex (left) and its associated 2-simplex facets (right)

permutation of $\vec{\Delta}$. The *twin* for some oriented simplex $\vec{\Delta}$ is defined as an oriented simplex $\vec{\Delta}'$ that is an *odd* permutation of $\vec{\Delta}$. For an oriented d-simplex $\vec{\Delta}$, its *faces* are the collection of all of its subsequences with the added condition that if the subsequence differs from $\vec{\Delta}$ by only one vertex and that vertex has an odd rank in $\vec{\Delta}$, that subsequence *must* be permuted by an *odd* permutation to be considered a face of $\vec{\Delta}$. The facets of $\vec{\Delta}$ are its faces of length d. A coface of $\vec{f}$ is an oriented simplex $\vec{\Delta}$ such that $\vec{f}$ is a face of $\vec{\Delta}$. A cofacet of $\vec{f}$ is a coface $\vec{\Delta}$ such that $\dim \vec{\Delta} = \dim \vec{f} + 1$. We define $\textbf{faces}\left(\vec{\Delta}\right)$, $\textbf{faces}_k\left(\vec{\Delta}\right)$, $\textbf{facets}\left(\vec{\Delta}\right)$, $\textbf{cofaces}\left(\vec{\Delta}\right)$, and $\textbf{cofacets}\left(\vec{\Delta}\right)$ for some oriented simplex $\vec{\Delta}$ similarly to the non-oriented variants. We define the intersection of two oriented simplices $\vec{\Delta}$ and $\vec{\Delta}'$, $\vec{\Delta} \cap \vec{\Delta}'$, is a set of the highest dimensional twin pairs $\{(\vec{f}, \vec{f}')\}$ where for each twin pair $(\vec{f}, \vec{f}')$, $\vec{f}$ is a face of $\vec{\Delta}$ and $\vec{f}'$ is a face of $\vec{\Delta}'$.

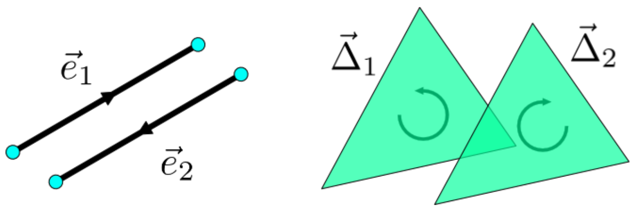

Figure 2.3: Example oriented twin 1-cells (left) and oriented twin 2-cells (right)

A *simplicial complex* $\mathcal{K}$ is such that for all simplices $\Delta \in \mathcal{K}$, $\textbf{faces}(\Delta) \subseteq \mathcal{K}$ and for every two simplices $\Delta, \Delta' \in \mathcal{K}$, their intersection $\Delta \cap \Delta'$ must be a face of both Δ and Δ'. Similarly, an *oriented* simplicial complex $\vec{\mathcal{K}}$ is such that for all oriented simplices $\vec{\Delta} \in \vec{\mathcal{K}}$, $\textbf{faces}\left(\vec{\Delta}\right) \subseteq \vec{\mathcal{K}}$ and for every two oriented simplices $\vec{\Delta}, \vec{\Delta}' \in \vec{\mathcal{K}}$, $\vec{f}$ is a face of $\vec{\Delta}$ and $\vec{f}'$ is a face of $\vec{\Delta}'$ for all $(\vec{f}, \vec{f}') \in \left(\vec{\Delta} \cap \vec{\Delta}'\right)$. A simplicial k-complex $\mathcal{K}$ is a complex where the highest dimension of any simplex $\Delta \in \mathcal{K}$ is k and a pure simplicial k-complex is a simplicial k-complex where all simplices with dimension less than k are the face of some k-simplex within the complex. For a simplicial k-complex, a *chamber* simplex is defined as any simplex with a dimension of k. Oriented simplicial complexes, pure oriented simplicial complexes, and their oriented chamber simplices are defined similarly.

The definitions above can readily be generalized to other classes of polytopes other than simplices, but simplices best represent our needs and so we will restrict ourselves to simplices going forward.

2.2 STRATIFIED SPACES AND THEIR GRAPHICAL REPRESENTATION

New developments within this work pertain to explicit handling of meshes represented by multiple manifolds. Being able to manage such meshes allow for more complicated physical modeling and in turn allow for doing a wider range of computational science. Going off of work in the previous section, a special interest is in having meshes represented as manifolds represented by oriented simplicial complexes.

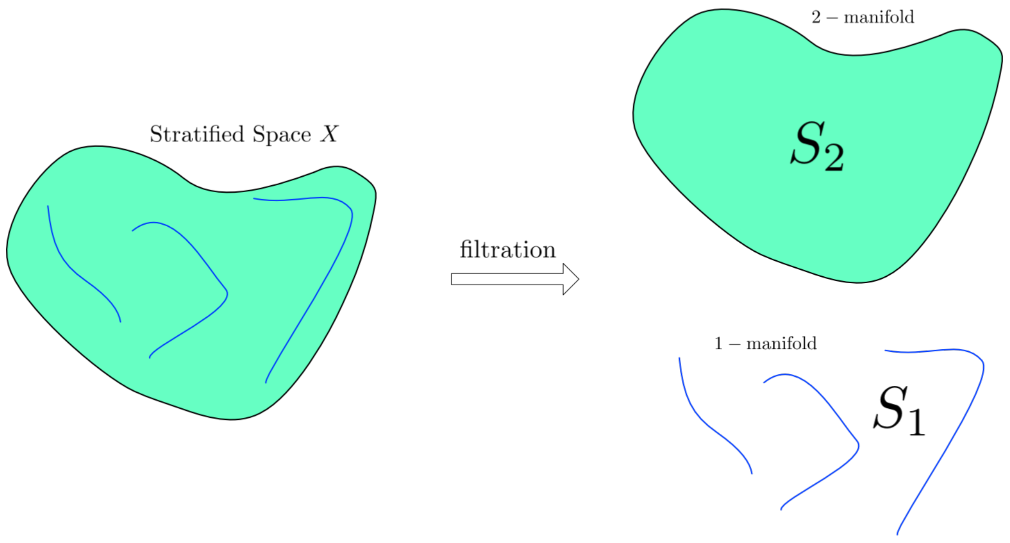

Figure 2.4: Example Stratified Space and Filtration into Manifolds

Recall that an n-manifold M is a topological space such that any point $p \in M$ has a neighborhood that is *homeomorphic* to $\mathbb{E}^n$. For our purposes, we define *stratified space* as a space X with a filtration

$$\emptyset = S_0 \subset S_1 \subset \cdots \subset S_n = X$$

where each S_k is a k-manifold and each k-stratum is S_k with its boundary. We construct meshes of objects that can be represented by this stratified space and, ultimately, as a set of overlapping manifolds with mixed dimensions. We assume for our purposes that each stratum is comprised of a single manifold, potentially represented as multiple disjoint manifolds. We then view meshes of this stratified space as a collection of mixed dimensional oriented manifolds, each k-dimensional manifold S_k represented as a *pure* oriented simplicial k-complex $\vec{M}_k$. The *codimension* of any simplex $\vec{\Delta} \in \vec{M}_k$ is equal to $\dim \vec{M}_k$ subtracted by $\dim \vec{\Delta}$. To ensure consistent orientation, each chamber of the complex is assigned the same orientation as the manifold that contains it. This implies that each stratum must be comprised of orientable manifolds, e.g. a Mbius strip cannot exist within a stratum.

We now describe our graphical representation of meshes representing stratified spaces, called the *stratified mesh*. The stratified mesh of an n-dimensional stratified space is a graph $G_n = (V_n, E_n)$ whose vertices are based on the chambers, and their associated facets, of the strata and whose edges encode adjacency information. Define the vertices to be $V_n = \left\{(k, \vec{\sigma}) \mid \vec{\sigma} \in \vec{M}_k\right\}$. The edges E_n are comprised of *three* classes of edges: *twin* edges, *facet* edges, and *shift* edges. A *facet* edge $\{(k, \vec{f}), (k, \vec{\Delta})\}$ is associated with an oriented k-manifold $\vec{M}_k$ and defined between two oriented simplices $\vec{f}, \vec{\Delta} \in \vec{M}_k$ when $\vec{f}$ is a facet of $\vec{\Delta}$. A *twin* edge $\{(k, \vec{\Delta}), (k, \vec{\Delta}')\}$ is defined between two oriented simplices $\vec{\Delta}, \vec{\Delta}' \in \vec{M}_k$ when $\vec{\Delta}$ is the twin of $\vec{\Delta}'$. A *shift* edge $\{(k-1, \vec{\Delta}), (k, \vec{\Delta}')\}$ is defined between oriented simplices $\vec{\Delta} \in \vec{M}_{k-1}$ and $\vec{\Delta}' \in \vec{M}_k$ when $\vec{\Delta}$, referred to as an *interstitial* simplex, is equivalent to $\vec{\Delta}'$, which is a *cointerstitial* simplex. E_n is the collection of all facet, twin, and shift edges found using $\{\vec{M}_k\}$.

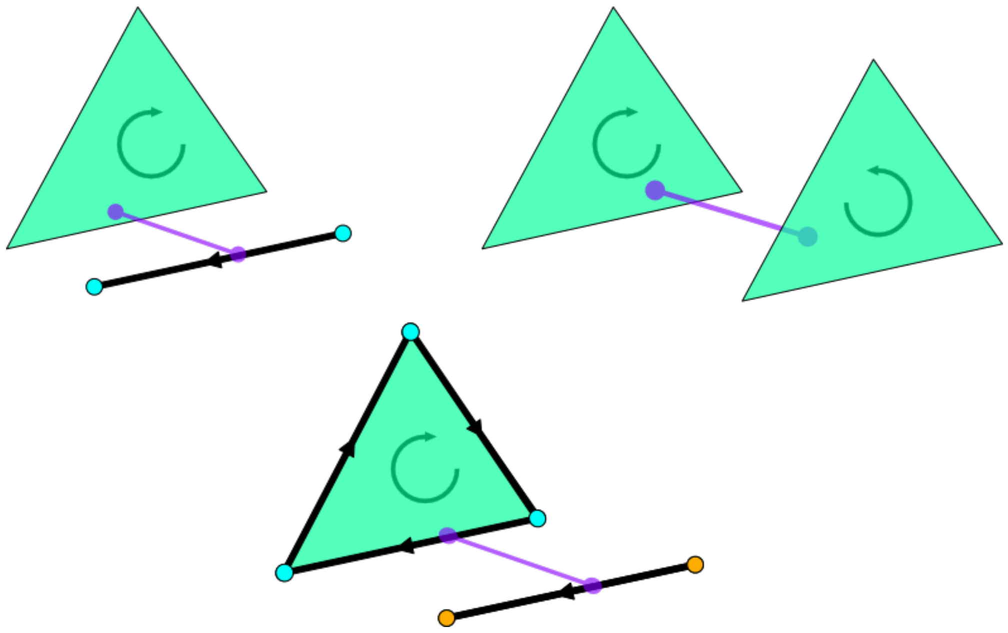

Figure 2.5: Example *facet* edge (top-left), example *twin* edge (top-right), and example *shift* edge (bottom-middle)

CHAPTER 3: MESHING DATA STRUCTURES AND ALGORITHMS

3.1 DATA STRUCTURE DESIGN FOR THE STRATIFIED MESH

In the previous section, we focused on constructing the *stratified mesh*, a graphical representation of the oriented manifolds within a stratified space. We now turn our attention to designing a data structure to represent this object. Within the context of spacetime discontinuous Galerkin methods, we have found no need to store oriented cells with codimension larger than 1 for a given oriented n-manifold, except that we *do* need to store all vertices of the oriented simplices. Thus, we ignore simplices in the stratified mesh G_n with codimension larger than 1 in their associated manifold, except vertices. Interestingly, this design can be viewed as a generalization of half-edge data structures [12] and shares similarity with the array-based half-facet (AHF) data structure designed at Sandia Labs [13]. With this clarification, we define a few intermediate data structures and then use them to construct a data structure that represents a stratified mesh.

Let us first assume we have some $\textsc{Set}$ data structure that allows common operations of $\textsc{Insert}$, $\textsc{Remove}$, $\textsc{Union}$, $\textsc{Intersection}$, and $\textsc{Size}$. We then fix a spatial dimension d such that our data structure represents a mesh of a stratified subset of $\mathbb{E}^d \times \mathbb{R}$. A $\textsc{Vertex}$ represents an point p in $\mathbb{E}^d \times \mathbb{R}$. Each $\textsc{Vertex}$ stores pointers to the simplices that contain p. An $\textsc{OrientedSimplex}_k$ O represents an oriented k-simplex with attributes in Table 3.1. Notice that within Table 3.1, we refer to a *global* versus a *non-global* manifold. A global manifold is the full k-manifold enclosing O while a non-global manifold could be any k-manifold that is a subset of the global manifold. The global and local IDs are important for indexing O in both the global and non-global manifolds simultaneously so we can easily move between them. We use this global and non-global terminology similarly for other data structures.

Now a $\textsc{Chamber}_k$ C represents an oriented k-simplex that is a chamber simplex of some oriented k-manifold. A $\textsc{Facet}_k$ F is an oriented k-simplex that is the facet of some $(k+1)$-simplex. Both $\textsc{Chamber}_k$ and $\textsc{Facet}_k$ inherit all the attributes of $\textsc{OrientedSimplex}_k$ with the specialization that F.manifold $= k + 1$ and C.manifold $= k$. We define attributes unique to $\textsc{Chamber}_k$ and $\textsc{Facet}_k$ in Tables 3.2 and 3.3. The C.facets and F.cofacet attributes represent the *facet* edges. The F.twin attribute represents the *twin* edges. The C.cointerstitial and F.interstitial attributes correspond to the *interstitial* edges.

Referring back to the $\textsc{Vertex}$ data structure, a $\textsc{Vertex}$ V will store its spacetime coordinate, a unique ID, and any $\textsc{Facet}$ and $\textsc{Chamber}$ data structures that contain it. These

OrientedSimplex$_k$	
ID$_g$	An ID set by its unique global manifold
ID$_l$	An ID set any non-global manifold
manifold	Dimension of manifold enclosing this simplex
vertices	A sequence of vertices whose ordering encodes the orientation
data	Arbitrary black-box satellite data

Table 3.1: Attributes of OrientedSimplex$_k$

Chamber$_k$	
facets$[1, \cdots, k+1]$	An array of $k+1$ Facet$_{k-1}$ type elements that correspond to its facets
cointerstitial	A Facet$_k$ instance that stores the cointerstitial simplex for this chamber; Nil if no cointerstitial simplex exists

Table 3.2: Attributes unique to Chamber$_k$

attributes are defined in Table 3.4. The attributes V.chambers and V.facets allow us to efficiently iterate through all incident simplices to a vertex, which becomes useful later on in spacetime meshing algorithms.

Our last intermediate data structure OrientedManifold$_k$ represents an oriented k-manifold with attributes in Table 3.5. The orientation of an OrientedManifold$_k$ M is enforced by requiring all chambers of this manifold to share the same orientation.

The main data structure of interest StratifiedMesh$_k$ corresponds to a k-dimensional stratified mesh and follows from the intermediate data structures defined previously. The attributes for this data structure are defined in Table 3.6.

Facet$_k$	
cofacet	A Chamber$_{k+1}$ type instance that stores the unique cofacet of this simplex; Nil if no cofacet exists
twin	A Facet$_k$ type instance that stores the twin of this simplex; Nil if no twin exists
interstitial	A Chamber$_k$ type instance that stores the interstitial simplex associated with this facet; Nil if no interstitial simplex exists

Table 3.3: Attributes unique to Facet$_k$

VERTEX	
ID$_g$	An ID set by a global stratified mesh
ID$_l$	An ID set by any non-global stratified mesh
point	A point value in $\mathbb{E}^d \times \mathbb{R}$
chambers$[1, \cdots, d+1]$	An array of $d+1$ SET elements that store CHAMBER type elements incident to this vertex; element k of the array corresponds to a set of CHAMBER$_k$ elements contained in the k-manifold of the strata
facets$[1, \cdots, d+1]$	An array of $d+1$ SET elements that store FACET type elements incident to this vertex; element k of the array corresponds to a set of FACET$_k$ elements contained in the k-manifold of the strata

Table 3.4: Attributes of VERTEX

ORIENTEDMANIFOLD$_k$	
orientation	An integer value of -1 or 1 representing the orientation.
chambers$[1, \cdots, n_c]$	An array of n_c CHAMBER$_k$ elements contained within the manifold
facets$[1, \cdots, n_f]$	An array of n_f FACET$_{k-1}$ elements contained within the manifold

Table 3.5: Attributes of ORIENTEDMANIFOLD$_k$

3.2 SUPPORTING ALGORITHMS

Given the data structures defined in the previous section, a collection of algorithms specific to them are defined. Algorithm 3.1 and 3.2 are procedures that return the dimension and codimension of any ORIENTEDSIMPLEX$_k$ instance, including CHAMBER$_k$ and FACET$_k$ since they inherit from ORIENTEDSIMPLEX$_k$. The algorithms for obtaining the dimension for ORIENTEDMANIFOLD$_k$ and STRATIFIEDMESH$_k$ are equivalent to Algorithm 3.1 after changing the input type accordingly.

Algorithm 3.1 Dimension of ORIENTEDSIMPLEX$_k$ O

```
1: procedure $O$.DIMENSION( )
2:     return k
3: end procedure
```

StratifiedMesh$_k$	
vertices$[1, \cdots, n_v]$	An array of n_v Vertex elements that comprise the mesh
manifolds$[1, \cdots, k]$	An array of size k comprised of OrientedManifold elements where element i is specifically OrientedManifold$_i$ type. Element i can be Nil for $i < k$ but element k must not be Nil

Table 3.6: Attributes of StratifiedMesh$_k$

Algorithm 3.2 Codimension of OrientedSimplex$_k$ O

```
1: procedure O.Codimension( )
2:     return manifold - O.Dimension()
3: end procedure
```

Algorithms 3.3 through 3.10 correspond to methods that will get or set state related to a Vertex instance. Algorithm 3.11 extracts all simplices incident to a vertex in a corresponding StratifiedMesh and returns this result in the form of a StratifiedMesh instance. Algorithm 3.12 tells us if some oriented simplex is incident to some vertex and this readily generalizes to Facet and Chamber instances. Algorithms 3.13 and 3.14 provide a way for us to add and remove chamber cells and their associated facets from an oriented mesh. Note that these algorithms assume the underlying array data structure has methods Exists, Remove, and Get-Open-ID. The method Exists returns a boolean value of True if the input value exists in the array, Remove removes an element from an array, and Get-Open-ID returns an index to an element of the array that is not currently being used and is available to be used by some value. Also note that Algorithm 3.13 is defined for the local ID but is similarly defined for the global ID.

Algorithms 3.15 and 3.16 represent methods that allow one to add or remove vertices from a StratifiedMesh$_k$ instance. Algorithms 3.17 and 3.18 allow one to add or remove chamber simplices, their associated facets, and any interstitial simplices from a StratifiedMesh$_k$ instance. The insertion algorithm for the chambers depends on the array of manifold types having a method Allocate that takes an input dimension k and allocates storage for the k-manifold spot. These algorithms form the basis for the key data structure operations.

Algorithm 3.3 Get spatial coordinate of VERTEX V

```
1: procedure $V$.SPATIAL-COORDINATE( )
2:     return $V$.point$[1, \cdots, d]$
3: end procedure
```

Algorithm 3.4 Get time coordinate of VERTEX V

```
1: procedure $V$.TIME-COORDINATE( )
2:     return $V$.point$[d+1]$
3: end procedure
```

Algorithm 3.5 Get set of CHAMBER$_k$ elements incident to VERTEX V

```
1: procedure $V$.INCIDENT-CHAMBERS$_k$( )
2:     return $V$.chambers[k]
3: end procedure
```

Algorithm 3.6 Get set of FACET$_k$ elements incident to VERTEX Vh

```
1: procedure $V$.INCIDENT-FACETS$_k$( )
2:     return $V$.facets[k+1]
3: end procedure
```

Algorithm 3.7 Insert CHAMBER$_k$ C instance into VERTEX V

```
1: procedure $V$.INSERT-CHAMBER$_k$($C$)
2:     $V$.chambers[k].INSERT($C$)
3: end procedure
```

Algorithm 3.8 Insert FACET$_k$ F instance into VERTEX V

```
1: procedure $V$.INSERT-FACET$_k$($F$)
2:     $V$.facets[k].INSERT($F$)
3: end procedure
```

Algorithm 3.9 Remove CHAMBER$_k$ C instance from VERTEX V

```
1: procedure $V$.REMOVE-CHAMBER$_k$($C$)
2:     $V$.chambers[k].REMOVE($C$)
3: end procedure
```

Algorithm 3.10 Remove $\textsc{Facet}_k$ F instance from $\textsc{Vertex}\,V$

```
1: procedure $V$.Remove-Facet$_k$($F$)
2:     $V$.facets[k].Remove($F$)
3: end procedure
```

Algorithm 3.11 Get star of $\textsc{Vertex}\,V$ in Stratified Mesh

```
1: procedure $V$.star( )
2:     StratifiedMesh$_d$ $S$                         ▷ init Stratified Mesh to store star of $V$
3:     $S$.Add-Vertex($V$)                              ▷ add vertex to the stratified mesh
4:     for Chamber$_d$ $C$ in $V$.chambers[d] do          ▷ insert $d$-dimensional chambers
5:         for Vertex $V'$ in $C$.vertices do             ▷ add vertices of chamber to mesh
6:             $S$.Add-Vertex($V'$)
7:         end for
8:         $S$.Insert-Chamber$_d$($C$)
9:     end for
10:    return $S$
11: end procedure
```

Algorithm 3.12 Check if oriented k-simplex O contains vertex V

```
1: procedure $O$.Contains-Vertex$_k$($V$)
2:     for Vertex $V'$ in $O$.vertices do
3:         if $V = V'$ then return True
4:         end if
5:     end for
6:     return False
7: end procedure
```

Algorithm 3.13 Add CHAMBER$_k$ C and its facets into ORIENTEDMANIFOLD$_k$ M

```
 1: procedure $M$.INSERT-LOCAL$_k$($C$)
 2:     if $\neg M$.chambers.EXISTS($C$) then            ▷ check if chamber $C$ is in array
 3:         $C$.ID$_l \leftarrow M$.chambers.GET-OPEN-ID()    ▷ get local ID for chamber from array
 4:         $M$.chambers[$C$.ID$_l$] $\leftarrow C$
 5:         for FACET$_{k-1}$ $F$ in $C$.facets do
 6:             $F$.ID$_l \leftarrow M$.facets.GET-OPEN-ID()      ▷ get local ID for facet from array
 7:             $M$.facets[$F$.ID$_l$] $\leftarrow F$
 8:         end for
 9:     end if
10: end procedure
```

Algorithm 3.14 Remove CHAMBER$_k$ C and its facets from ORIENTEDMANIFOLD$_k$ M

```
1: procedure $M$.REMOVE$_k$($C$)
2:     if $M$.chambers.EXISTS($C$) then                  ▷ check if chamber $C$ is in array
3:         $M$.chambers.REMOVE($C$)
4:         for FACET$_{k-1}$ $F$ in $C$.facets do
5:             $M$.facets.REMOVE($F$)
6:         end for
7:     end if
8: end procedure
```

Algorithm 3.15 Add VERTEX V to Stratified Mesh S

```
1: procedure $S$.ADD-VERTEX-LOCAL($V$)
2:     if $\neg S$.vertices.EXIST($V$) then
3:         $V$.ID$_l \leftarrow S$.vertices.GET-OPEN-ID()       ▷ get local ID for vertex from array
4:     end if
5: end procedure
```

Algorithm 3.16 Remove VERTEX from STRATIFIEDMESH S

```
1: procedure $S$.REMOVE-VERTEX($V$)
2:     if $S$.vertices.EXIST($V$) then
3:         $S$.vertices.REMOVE($V$)
4:     end if
5: end procedure
```

Algorithm 3.17 Insert CHAMBER$_k$ and its facets into STRATIFIEDMESH$_k$ S

```
procedure $S$.INSERT$_k$($C$)
    if $S$.manifold[k] = NIL then                          ▷ allocate $k$-manifold if needed
        $S$.manifold.ALLOCATE($k$)
    end if
    $S$.manifold[k].INSERT$_k$($C$)          ▷ insert chamber and its facets into the $k$-manifold
    for FACET$_{k-1}$ $F$ in $C$.facets do
        if $f$.interstitial $\neq$ NIL then   ▷ recursively insert interstitial simplex if not NIL
            $S$.INSERT$_{k-1}$($f$.interstitial)
        end if
    end for
end procedure
```

Algorithm 3.18 Remove CHAMBER$_k$ and its facets from STRATIFIEDMESH$_k$ S

```
procedure $S$.REMOVE$_k$($C$)
    if $S$.manifold[k] = NIL then         ▷ return since $C$ can not reside in this manifold
        return
    end if
    $S$.manifold[k].REMOVE$_k$($C$)         ▷ remove chamber and its facets in the $k$-manifold
    for FACET$_{k-1}$ $F$ in $C$.facets do
        if $f$.interstitial $\neq$ NIL then ▷ recursively remove interstitial simplex if not NIL
            $S$.REMOVE$_{k-1}$($f$.interstitial)
        end if
    end for
end procedure
```

3.3 STRATIFIED MESHES AND THE TENTPITCHER ALGORITHM

3.3.1 Fundamental Concepts of the TentPitcher Algorithm

The TentPitcher algorithm is a spacetime meshing algorithm fundamental to modern spacetime discontinuous Galerkin techniques used to solve systems of Hyperbolic Partial Differential Equations (PDEs). This algorithm exploits the property of hyperbolic PDEs that information propagates through spacetime at finite rates tied to the characteristics of the hyperbolic system. This exploitation allows for a time and space efficient algorithm for constructing spacetime simplices that can then be used to efficiently solve the hyperbolic system of PDEs.

The characteristics of the hyperbolic system induce the *cone constraint* [4] for the facets of all the spacetime simplices. The modern TentPitcher algorithm works with a space mesh $M \subset \mathbb{E}^d$ representing a single manifold called the *ground mesh*. The algorithm constructs spacetime simplices using simplices in the ground mesh that satisfy the cone constraint and ensure continuous construction of spacetime simplices up to any desired time [4]. Ever since the theoretical work and algorithm that allowed for spacetime meshing over arbitrary spatial domains, further work has been done that generalizes the technique for adaptive operations such as moving vertices or performing adaptive refinement and coarsening [5, 6].

The TentPitcher algorithm begins by attaching time values to each vertex in the ground mesh, creating a d-manifold embedded in $\mathbb{E}^d \times \mathbb{R}$ referred to as the *Front*. For any vertex v in the Front, the *Footprint* of this vertex has historically been the star of v, where $\text{star}(v) = \{\Delta \in M \mid \Delta \cap v \neq \emptyset\}$. Within this work, we redefine the footprint to be what we internally call the *extended* footprint. This extended footprint is defined below in (3.1) and will be what we refer to as a footprint going forward. With this new definition, a footprint about some vertex v is the collection of all simplices that are incident to the 1-simplices that are faces of the simplices found in the star of v. The redefinition is motivated by parallel implementations that require more mesh information so that various adaptive operations are better done in parallel. Parallel generalization of the adaptive operations is heavily tied to *lazy refinement* and *lazy clean-up* found in [6] and are a key motivator for the new footprint definition.

$$\text{Footprint}(v) = \bigcup_{f \in \text{star}(v)} \bigcup_{c \in \mathbf{faces}_1(f)} \text{star}(c) \tag{3.1}$$

The Front defined above is also be viewed as the graph of some continuous time function

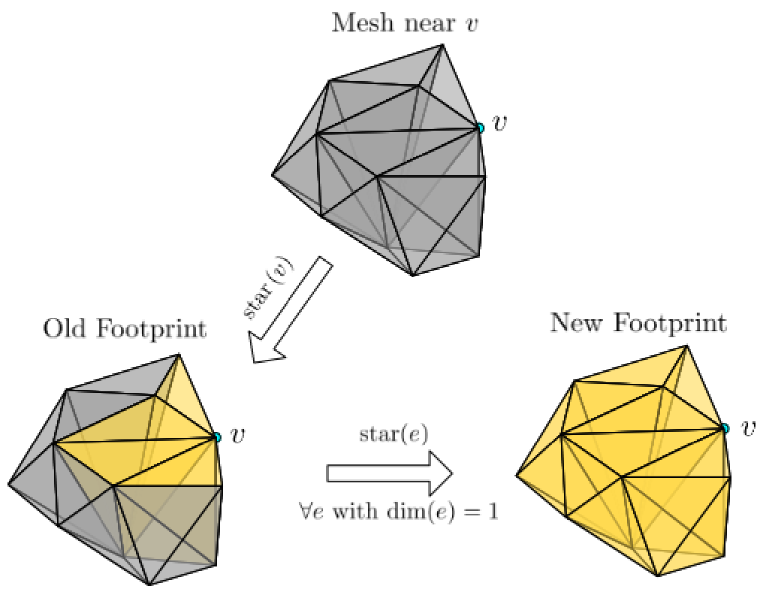

Figure 3.1: Differences between original Footprint and *extended* Footprint about a vertex v for a sample mesh. Gray cells are not in the Footprint, Gold cells are.

$t : \mathbb{E}^d \to \mathbb{R}$ that maps spatial coordinates to a time value in a piecewise affine manner using the time values associated with the vertices of the simplices [4]. If one restricts the time function $t(\boldsymbol{x})$ to some simplex, then $t(\boldsymbol{x})$ is affine. It is convenient to represent a facet of the Front as a simplex represented by the tuple $(\Delta, \boldsymbol{t})$, where $\Delta = \{v_0, \cdots, v_d\}$ is some d-simplex and $\boldsymbol{t} = (t_0, \cdots, t_d)$ are the time values associated with each vertex [7]. The restriction of the time function to the spacetime simplex $(\Delta, \boldsymbol{t})$ is $t(\Delta, \boldsymbol{t}) : \mathbb{E}^d \to \mathbb{R}$ and the simplex is *causal* if the restricted time function satisfies $\|\nabla_x t(\Delta, \boldsymbol{t})\|_2 \leq \frac{1}{s_{\max}}$ for the local maximum wavespeed $s_{\max}$. From [7], we define $t_{(i)}$ as the element of $\boldsymbol{t}$ with rank i, meaning that $t_{(i)}$ is greater than or equal to i elements of $\boldsymbol{t}$, making $t_{(0)}$ the minimum time value. Further, the binary function $\boldsymbol{t} \uparrow x$ is equivalent to updating the smallest vertex of $\boldsymbol{t}$ by doing $t_{(0)} \leftarrow \max\{t_{(0)}, x\}$ and the binary function $\boldsymbol{t} \Uparrow x$ similarly updates *all* time values to be $t_{(i)} \leftarrow \max\{t_{(i)}, x\}$.

The abstract TENTPITCHER algorithm given in [7] is restated in Algorithm 3.19. In Algorithm 3.19, M is the ground mesh, $\boldsymbol{S}$ is a time vector indexed by the vertices of M, $\delta \in \left(0, \frac{1}{2}\right)$ is a fixed parameter, and T_f is a time that we request the TENTPITCHER algorithm propagate the Front past. This algorithm is based on work from [4] and [7] that allows for spacetime meshing over arbitrary spatial meshes. For each simplex in the Footprint of v, we first find a time t^*_Δ as large as possible that we can pitch v to such that $(\Delta, \boldsymbol{t} \uparrow t^*_\Delta)$ is *Valid*. As discussed in [7], a valid time vector $\boldsymbol{t}$ will be an element of a set Valid(Δ) which has the properties that Valid(Δ) is open, convex, any $(\Delta, \boldsymbol{t})$ where $\boldsymbol{t} \in$ Valid(Δ) is causal, and for any $\boldsymbol{t} \in$ Valid(Δ), $(\boldsymbol{t} \Uparrow t_i) \in$ Valid(Δ) for all i. It is shown that a construction

Algorithm 3.19 TentPitcher

```
1: procedure TentPitcher($M, \boldsymbol{S}, \delta, T_f$)
2:     $\boldsymbol{T} \leftarrow \boldsymbol{S}$
3:     while $\min \boldsymbol{T} \leq T_f$ do
4:         $v \leftarrow$ Choose arbitrary local minimum vertex of M
5:         $t_{\text{new}} \leftarrow \infty$
6:         for each simplex $(\Delta, \boldsymbol{t})$ in star $(v)$ do
7:             $t^*_\Delta \leftarrow \sup\{x \mid (\Delta, \boldsymbol{t} \uparrow x)$ is valid$\}$
8:             $\rho \leftarrow$ Choose arbitrary value within $(\delta, 1-\delta)$
9:             $t_\Delta \leftarrow \rho t^*_\Delta + (1-\rho) t_{(1)}$
10:            $t_{\text{new}} \leftarrow \min\{t_{\text{new}}, t_\Delta\}$
11:        end for
12:        $\boldsymbol{T}[v] \leftarrow t_{\text{new}}$
13:    end while
14:    return
15: end procedure
```

for Valid(Δ) that satisfies these properties is Valid(Δ) $= \{\boldsymbol{t} \mid (\boldsymbol{t} \Uparrow x) \in \text{Causal}(\Delta)\, \forall x \in \mathbb{R}\}$, where Causal($\Delta$) is the set of all time vectors $\boldsymbol{t}$ such that $(\Delta, \boldsymbol{t})$ is causal.

After finding t^*_Δ, we then perform an arbitrary convex combination between t^*_Δ and $t_{(1)}$ to produce a new time value t_Δ that ensures progress relative to $t_{(1)}$. By updating t_{new} to the value of $\min\{t_{\text{new}}, t_\Delta\}$, we ensure that t_{new} is valid for all incident simplices to v.

When a single pitch with the TentPitcher algorithm is performed, we construct spacetime simplices in between the old footprint incident on v and the new footprint incident to the pitched vertex v'. This collection of spacetime simplices in between the old and new Front fragments is called a *Patch.* A patch is a $(d+1)$-manifold that represents a piece of spacetime where we wish to find solutions to the hyperbolic PDEs. Facets of the $(d+1)$-simplices incident to the old footprint are denoted *inflow* facets while the facets incident to the new footprint are denoted *outflow* facets. Spacetime simplices incident to and below the inflow facets are called *predecessor cells* and are coupled to the spacetime simplices within the patch due to the cone constraint. The TentPitcher algorithm is used to construct patches in spacetime such that the predecessor cells are always comprised of *known* solution information, which we call *inactive* cells. This feature allows us to solve for the solution field within the patch, comprised of *active* simplices, using only local information.

Using the TentPitcher algorithm, we construct a patch of simplicies to solve in $O(1)$ time with $O(1)$ memory cost. We then solve the system of equations for the degrees of freedom of the patch using only local information. Performing this solution procedure for each possible patch until we move the Front past some desired time T_f results in an $O(n)$

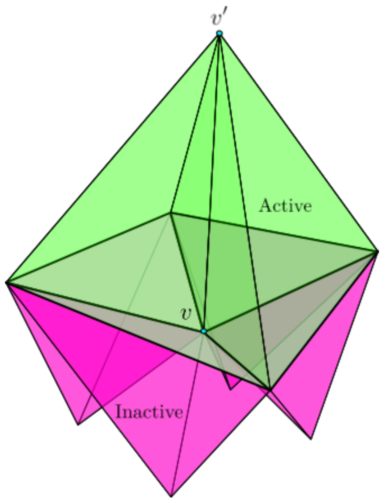

Figure 3.2: Example Patch created around vertex v with Active and Inactive cells

algorithm, where n is the number of $(d+1)$-simplices that make up the spacetime mesh. Once an inactive spacetime simplex is not incident to any facet on the Front, we can save and delete (or recycle) its information and maintain a spacetime mesh proportional in size to the input spatial mesh.

3.3.2 Historical Developments of TentPitcher

The TentPitcher algorithm is the culmination of an effort spanning almost two decades that has produced numerous theoretical and practical results in the context of meshing a single manifold through spacetime. Early work in the TentPitcher algorithm by ngr and Sheffer [3] found one could simply move a vertex in time as far as possible while still being causal as long as the angles of the spatial simplices were acute. Using this, the TentPitcher algorithm was born and capable of constructing unstructure spacetime meshes of the entire spacetime domain using an iterative procedure. Work by Erickson et. al. [4] devised an improvement to the TentPitcher algorithm and generated various theoretical guarantees that ensured arbitrary spatial simplices could be meshed in spacetime, even for higher dimensions.

Later, developments in [5, 14, 6, 7] made great strides towards fully adaptive spacetime algorithms that included spacetime smoothing, pitching tents in specified spacetime directions, refinement and coarsening through $3d \times$ time, and the ability to adaptively mesh and work in nonlinear and anisotropic media. In $1d \times$ time and $2d \times$ time, adaptive spacetime meshing is solved, meaning it is known how to set up such operations such that arbitrary progress in time can occur. In $3d \times$ time, there is still a question about finding a policy that can use adaptive refinement and coarsening techniques together and provably leave the mesh in a state where one can make arbitrary progress in time [7].

The use of the TentPitcher algorithm alongside Spacetime Discontinuous Galerkin methods developed by Haber and others [15, 16, 17, 18] has been used in a wide range of computational science applications, some examples being [19, 20, 21, 22, 23, 24, 25, 26]. For a more detailed summary about historical work on the TentPitcher algorithm, the reader can refer to [7].

3.3.3 TentPitcher and the StratifiedMesh

In this section, we generalize the TentPitcher algorithm to objects represented by a StratifiedMesh. Fix a spatial dimension d such that our mesh is a stratified subset of $\mathbb{E}^d \times \mathbb{R}$. A ground mesh, the Front, and a footprint are generalized to a d-dimensional stratified mesh, allowing us to model them with a StratifiedMesh_d. Similarly, a patch is generalized to a $(d+1)$-dimensional stratified mesh, allowing us to model it as a $\text{StratifiedMesh}_{d+1}$. With these representations, we naively modify the TentPitcher algorithm to find valid pitches for some vertex v using all simplices incident to v across *all* the manifolds. This naive algorithm is defined abstractly in Algorithm 3.20 using data structures from earlier.

Algorithm 3.20 Naive Stratified TentPitcher

```
1: procedure Naive-Stratified-TentPitcher($M, \boldsymbol{S}, \delta, T_f$)
2:     $\boldsymbol{T} \leftarrow \boldsymbol{S}$
3:     while $\min \boldsymbol{T} \leq T_f$ do
4:         $v \leftarrow$ Choose arbitrary local minimum vertex of M
5:         $t_{\text{new}} \leftarrow \infty$
6:         for $k$ in $d, d-1, \cdots$ do
7:             for each $\text{Chamber}_k$ $(\Delta, \boldsymbol{t})$ in $v.\text{Incident-Chambers}_k()$ do
8:                 $t^*_\Delta \leftarrow \sup\{x \mid (\Delta, \boldsymbol{t} \uparrow x) \text{ is valid}\}$
9:                 $\rho \leftarrow$ Choose arbitrary value within $(\delta, 1-\delta)$
10:                $t_\Delta \leftarrow \rho t^*_\Delta + (1-\rho) t_{(1)}$
11:                $t_{\text{new}} \leftarrow \min\{t_{\text{new}}, t_\Delta\}$
12:            end for
13:        end for
14:        $\boldsymbol{T}[v] \leftarrow t_{\text{new}}$
15:    end while
16:    return
17: end procedure
```

Algorithm 3.20 can be improved by first recalling from [4] that one can parametrize the gradient of the time function, restricted to some simplex $(\Delta, \boldsymbol{t})$, to be

$$\nabla_x t = \nabla_x t|_{f_i} + \hat{\boldsymbol{n}} \frac{\partial t}{\partial \hat{\boldsymbol{n}}} \tag{3.2}$$

where $\nabla_x t|_{f_i}$ is the gradient restricted to some facet $(f_i, \boldsymbol{t}_i)$ of $(\Delta, \boldsymbol{t})$ and $\frac{\partial t}{\partial \hat{\boldsymbol{n}}}$ is the directed derivative of the time function along a direction $\hat{\boldsymbol{n}}$ orthogonal to f_i. Since the TENT-PITCHER chooses a tentpole such that $(\Delta, \boldsymbol{t})$ is causal, it will find a tentpole such that $\|\nabla_x t\|_2 \leq \frac{1}{s_{\max}}$ is satisfied. If $(\Delta, \boldsymbol{t})$ is causal, its facet $(f_i, \boldsymbol{t}_i)$ is causal.

Proof.

$$\begin{aligned}
\left(\frac{1}{s_{\max}}\right)^2 &\geq \|\nabla_x t\|_2^2 \\
&= \left\| \nabla_x t|_{f_i} + \hat{\boldsymbol{n}} \frac{\partial t}{\partial \hat{\boldsymbol{n}}} \right\|_2^2 \\
&= \left\| \nabla_x t|_{f_i} \right\|_2^2 + \left\| \hat{\boldsymbol{n}} \frac{\partial t}{\partial \hat{\boldsymbol{n}}} \right\|_2^2 \\
&\geq \left\| \nabla_x t|_{f_i} \right\|_2^2 \\
\implies \quad \frac{1}{s_{\max}} &\geq \left\| \nabla_x t|_{f_i} \right\|_2
\end{aligned} \tag{3.3}$$

The result of (3.3) applies recursively to the facets of $(f_i, \boldsymbol{t}_i)$, so inductively this implies that all faces of Δ will be causal with respect to time vector $\boldsymbol{t}$ when $(\Delta, \boldsymbol{t})$ is causal. For a time vector $\boldsymbol{t}$ to be valid for some simplex Δ, we also require that $\boldsymbol{t}$ is such that $\forall u \in \mathbb{R}$, $(\boldsymbol{t} \Uparrow u)$ is causal with respect to Δ. Given we find $\boldsymbol{t}$ such that $(\Delta, \boldsymbol{t})$ is valid, we know that $\hat{\boldsymbol{t}}_u = (\boldsymbol{t} \Uparrow u)$ is causal with respect to Δ for any $u \in \mathbb{R}$. It follows from (3.3) that the faces of Δ are causal with respect to $\hat{\boldsymbol{t}}_u$ for any $u \in \mathbb{R}$, implying that the faces of Δ are valid with respect to the choice of $\boldsymbol{t}$. With that, we state an algorithm that highly resembles the original abstract TENTPITCHER, as seen in Algorithm 3.21. Since Algorithm 3.21 is effectively equivalent to the baseline TENTPITCHER algorithm, it carries the same theoretical guarantees.

Now for the TENTPITCHER implementation used to produce the later results, the algorithm is somewhat modified but is based on the respective works of Thite and Mont [6, 7]. Mont showed that the set of valid time vectors $\boldsymbol{t}$ is convex, implying the feasibility of a binary search styled algorithm to find a valid time vector [7]. Thite showed that it is possible to both pitch tents along some specified spacetime direction and to pitch tents in such

Algorithm 3.21 Stratified TENTPITCHER

```
procedure STRATIFIED-TENTPITCHER($M, \boldsymbol{S}, \delta, T_f$)
    $\boldsymbol{T} \leftarrow \boldsymbol{S}$
    while $\min \boldsymbol{T} \leq T_f$ do
        $v \leftarrow$ Choose arbitrary local minimum vertex of M
        $t_{\text{new}} \leftarrow \infty$
        for each CHAMBER$_d$ $(\Delta, \boldsymbol{t})$ in $v$.INCIDENT-CHAMBERS$_d$() do
            $t^*_\Delta \leftarrow \sup\{x \mid (\Delta, \boldsymbol{t} \uparrow x) \text{ is valid}\}$
            $\rho \leftarrow$ Choose arbitrary value within $(\delta, 1-\delta)$
            $t_\Delta \leftarrow \rho t^*_\Delta + (1-\rho)t_{(1)}$
            $t_{\text{new}} \leftarrow \min\{t_{\text{new}}, t_\Delta\}$
        end for
        $\boldsymbol{T}[v] \leftarrow t_{\text{new}}$
    end while
    return
end procedure
```

that the tentpole vertex moves towards some desired smoothed spatial coordinate [6]. A generalization of the tentpole vertex location is defined as the parametrized function

$$v'(s) = u + \hat{\boldsymbol{d}}s \tag{3.4}$$

where $v'(s)$ is the tentpole vertex as a function of parameter s, u is some base spacetime point, and $\hat{\boldsymbol{d}}$ is some spacetime direction. Specializing to the case of a tilted tentpole, we have that $u = v$ and that $\hat{\boldsymbol{d}}$ is specified as an input direction with limitations found in [6], where v is the vertex the footprint is based around. In the case of a smoothing operation, u is equal to the smoothed spatial location with the same time component as v and $\hat{\boldsymbol{d}} = (0,0,0,1)$. A typical TENTPITCHER operation specializes with $u = v$ and $\hat{\boldsymbol{d}} = (0,0,0,1)$. Viewing the new tentpole vertex v' in this parametric way generalizes the operations we already know. This generic form is solved using a binary search for a valid parameter s since these three cases have the theoretical backing that they need to ensure they will work. As of now, there are no guarantees about what will happen for an arbitrary $(u, \hat{\boldsymbol{d}})$ combination.

CHAPTER 4: SOFTWARE ARCHITECTURE AND DESIGN

4.1 AN EYE TOWARDS HIGH PERFORMANCE COMPUTING

With the data structures and algorithms of interest defined in the previous chapter, it becomes important to consider what context the implementations will have to work in. Within my group, serial codes have been the historic norm but there has been a shift in the recent years to develop scalable parallel codes that can work on shared memory and distributed memory systems. The desire to run Spacetime Discontinuous Galerkin codes with the TentPitcher algorithm require architecting the software with this in mind.

The abstract design that has been arrived at generally consists of the Front mesh being represented in a *global* manner and the footprints and patches represented in a *local* manner. *Global* here means that the Front mesh will be accessible by any thread or process involved in the implementation, while *local* means those objects will be managed by an individual thread or process. This distinction has a heavy implication on how the implementation of the various objects should be tailored so they can be optimal in terms of their run-time performance. Clearly, the Front will want to enforce some sort of locking behavior that will resist race conditions. This behavior is important to have since to scale, different threads/processes will need to be able to communicate with the Front data structure concurrently, whether it is to update the Front or extra footprints. This implies the use of internal data structures that can function concurrently with different threads or processes trying to extract information from the Front or update it.

Further, since footprints and patches will live on a given thread or process, such objects will not be working with much data since they will only need to manage a small neighborhood of data around some vertex. In the shared memory setting, it is also advantageous to copy this small amount of data into the associated thread so that false sharing between different threads can be avoided. These requirements imply that the data structures representing a footprint and patch should consider allocation on the stack or allocating a piece of the heap for use only within the associated thread or process.

Clearly, the Front requires different internal storage mechanisms than the footprints and patches, even though all of them are represented as a . However, to maximize code reuse, it is advantageous to not jump into developing a global and local version of the . Instead, this design dilemma implies the use of generic programming techniques to help generalize our data structures to handle the different requirements that exist for the project.

4.2 MODERN C++ AND GENERICS

In the previous chapter, the design of a k data structure and its more primitive data structures was designed that could be used to represent the information of interest within a k-dimensional Primary Mesh object. One can carry that parametrization into the implementation by using a generic driven approach. As discussed above, ensuring the data structures and algorithms can be used in a scalable manner when placed into a parallel environment, there is again a place for using generics when it comes to choosing appropriate containers for different data structures.

Due to the requirements of the project and other performance benefits, implementing a generic codebase using Template Metaprogramming in C++ is a natural move. By choosing such an approach, one immediate benefit is that the algorithms mentioned earlier can be specified at compile time relative to k, which will appear as a compile-time constant. This fact allows for more optimization opportunities by the compiler, everything from loop unrolling to appropriate function inlining. Another benefit is that generics can help with avoiding conditional branching since algorithm implementations will be fixed at run-time for the chosen k instead of needing to test the value of k at run-time and deciding which algorithm to use after. Branch prediction for the compiler becomes trivial, again leading to speed-ups.

The meshing data structures designed earlier in this thesis are the primary components taking advantage of all the generic programming, though certainly not the only ones. Generics have also been used to design a collection of objects we refer to as *Handles*, essentially light-weight data structures that hold on to a pointer to some object. The generic implementation not only allows for holding onto a generic pointer type, but it also allows for choosing whether the handle should *own* the data it points to or just be able to keep a reference to it. Using metaprogramming techniques allow this behavior to be specified in a very flexible way. Generics have also been used to design other generic data structures, like one used for doing *Object Pooling* to minimize dynamic allocation costs.

The main uses of generics are when it comes to implementing the meshing data structures. Metaprogramming techniques are used to not only parametrize meshing data structures with respect to their dimension k, but they are also used to make various functionality and type specifications decided at compile-time through the use of *Traits* and *Policy Classes* that allow for better compiler optimizations. Within the implemented software, Traits in particular are crucial. Traits bundle together a collection of types and potentially other applicable information [27, 28, 29] into a class that can modify all these types and information depending on the type of class passed in to the template parameter for the Traits class. Traits are used within the software to select different containers to use based on whether a data structure instance is marked at compile-time as *Global* or *Local*. Being able to parametrize the data structures in terms of containers has made the data structures automatically capable of scaling to parallel contexts by choosing an appropriate container that fits those requirements. This means that the Front can be specialized with an appropriate concurrent data structure while the footprints and patches can be specialized with faster, lower memory cost containers.

Traits are also used to specify whether it is appropriate to use a local indexing scheme or a global one and even help to decide what satellite data the data structures will carry depending on their dimensions and the simulation type being run. This is a useful feature since it effectively parametrizes the data structures with respect to the simulation one wishes to run, which makes sense to do. *Policy Classes* are used along with class composition to customize various behaviors with minimal coupling. These classes are used to implement a set of orthogonal functionality that can be used to build a variety of unique behaviors without running into the typical *Curse of Dimensionality* problem by explicitly implementing a class that implements every combination. These policy classes are used within meshing software to make it possible to swap certain implementations out in exchange for another that might be better, without requiring a change to anything but a template parameter. For those interested in learning more about these techniques and more, refer to [27, 28, 29].

4.3 VISUALIZATION

With the main C++ software written in a generic fashion, as outlined above, a variety of sample meshes have been constructed and used to test the generic TentPitcher algorithm implementation for a variety of cases. $2d$ meshes are constructed using the software named Triangle [30] and the $3d$ meshes are constructed using the software named TetGen [31]. Both software tools have been designed by experts in the field of meshing and work with input and output meshing files that are similar and simple to work with. Visualizations presented below were made using Matlab.

In Figure 4.1, the software tests the $3d\times$time capability of the generic codebase on a socket model, a sample model that comes with TetGen. This sample problem was used to validate that the TentPitcher code could push the Front past the time threshold of $T_f = 10$ seconds. The color field in the Front snapshots represents the time function $t : \mathbb{E}^d \times \mathbb{R} \to \mathbb{R}$ that the Front is associated with. As is visible in the color mapping, at just 500 pitches there is a visible subset of the mesh that has already pitched close to the time threshold T_f, while most of the mesh still has a minimal time value. This behavior is certainly expected as the TentPitcher algorithm is free to advance different portions of the Front at different rates. As the number of pitches increase, however, the time values across the Front begin to converge towards T_f, showing progress is being made as expected.

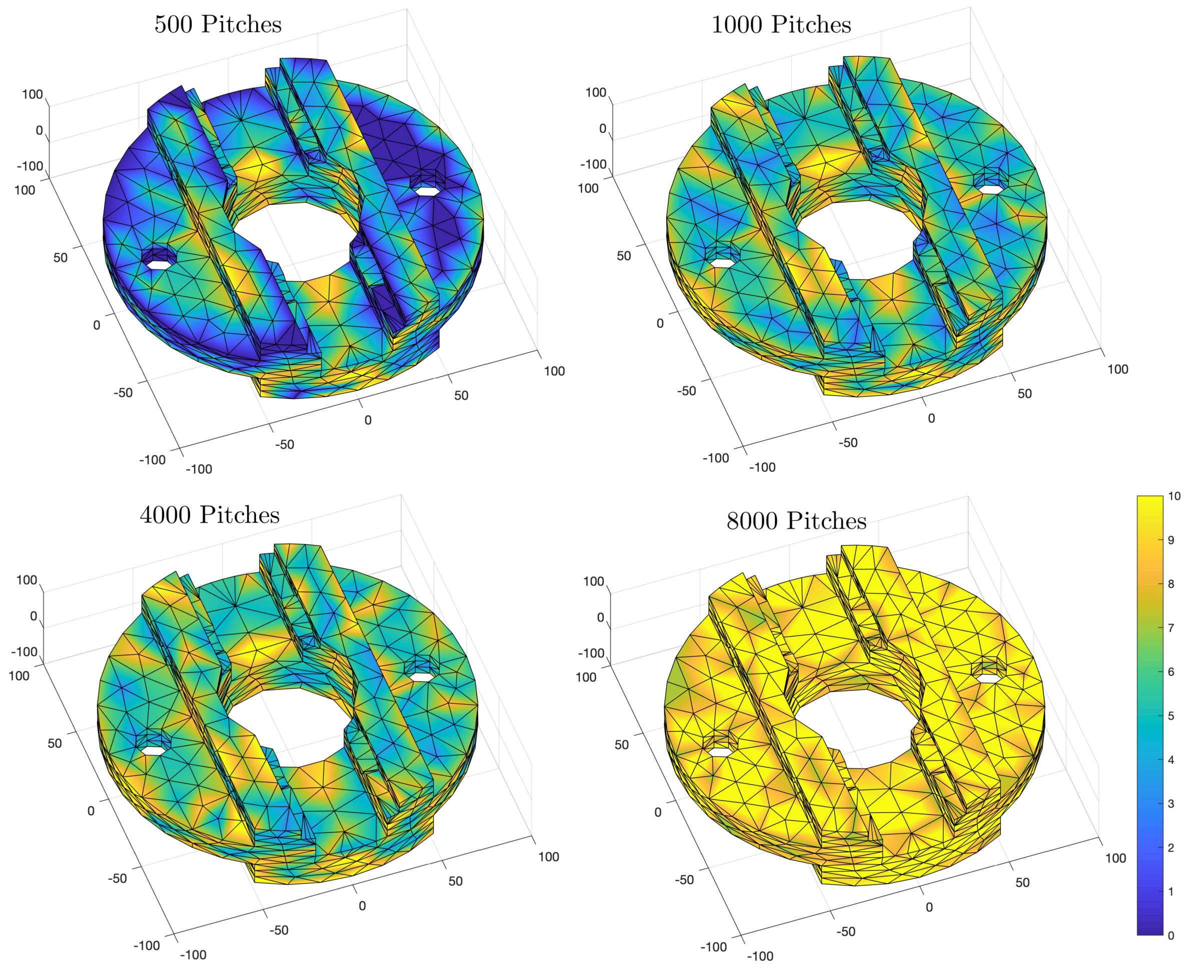

Figure 4.1: Pitching a socket in $3d \times$ time with color field representing time progress towards a goal of 10 seconds. TentPitcher took 9500 pitches to exceed the threshold time of 10 seconds.

Figure 4.2 provides a minimal test showing that we can smooth a simple simplicial complex in $3d \times$ time so that a mesh with extremely thin cells can have the mesh quality improved and in turn improve the rate at which TentPitcher can make progress through time. As one can see in the figure, the problematic vertex was moved to the center using smoothing operations after being placed initially adjacent to the vertex located at the bottom left corner of the box. The smoothing operation greatly improved the mesh quality and led to the TentPitcher algorithm being able to pitch further in time.

The last example in Figure 4.3 displays a visual for a $2d \times$ time example where the mesh is comprised of multiple manifolds. In this case, the ground mesh is a circle with a 1-manifold

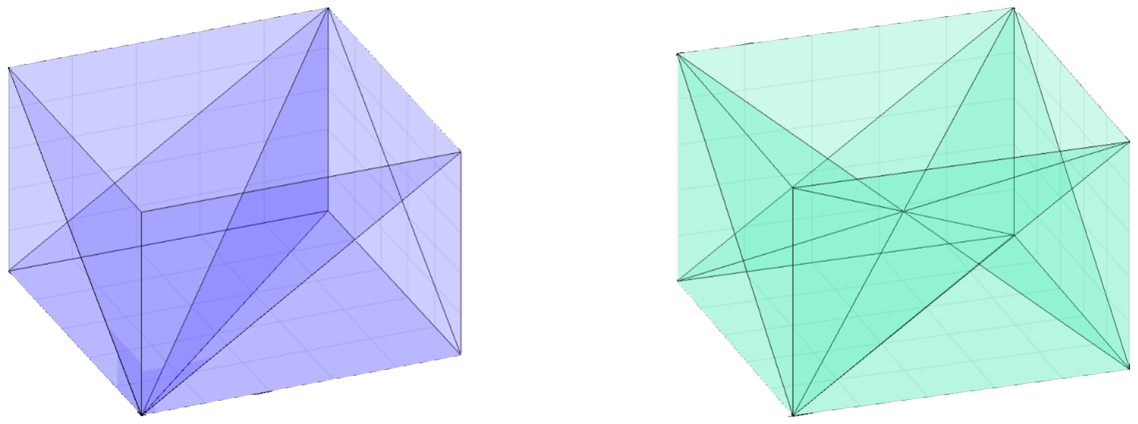

Figure 4.2: Basic spacetime smoothing test in $3d \times$ time. The left mesh is the initial spatial mesh with extremely thin elements near the bottom left vertex of the box. The right mesh is later in time after performing spacetime smoothing and improving the mesh quality.

ring embedded around the center. To visualize the progress of the 1-manifold portion of the Front, visualization of the 2-manifold portion of the front stopped after some time threshold. This test case allows the use of smoothing, whenever possible, to improve mesh quality as progress is made in time. Going off of the visual, the generic TENTPITCHER algorithm implemented is appears capable of meshing the stratified mesh through spacetime.

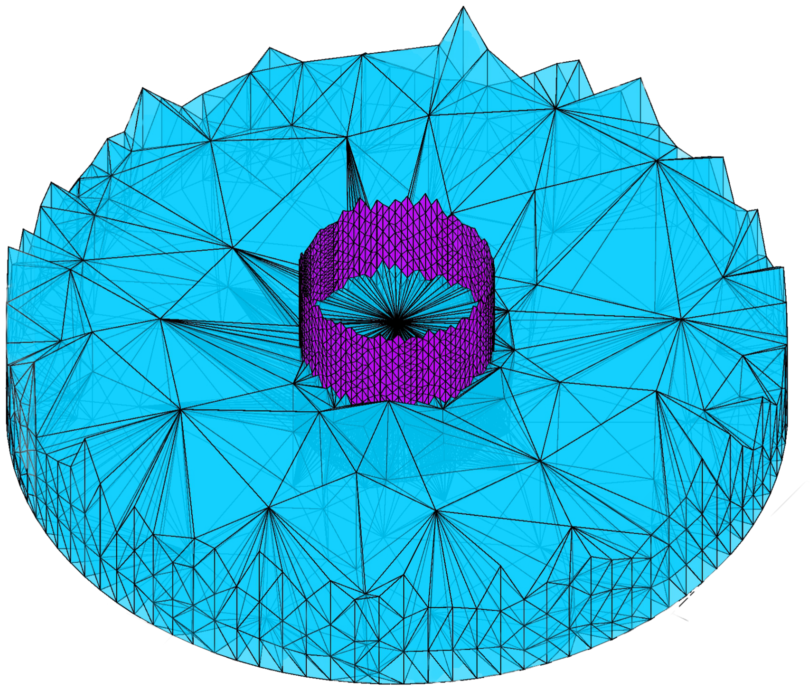

Figure 4.3: Pitching stratified spatial mesh with a 2-manifold and a 1-manifold using smoothing where ever possible. The spacetime cells associated with the 2-manifold are removed after some point to see progress of the 1-manifold in time.

REFERENCES

[1] R. J. Leveque, *Finite Volume Methods for Hyperbolic Problems*, ser. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002.

[2] M. T. Heath, *Scientific Computing: An Introductory Survey*, ser. McGraw-Hill Series in Computer Science. The McGraw-Hill Companies, Inc., 1997.

[3] A. ngr and A. Sheffer, "Pitching tents in space-time: Mesh generation for discontinuous galerkin method," *Int. J. Foundations of Computer Science*, vol. 13(2), pp. 201–221, 2001.

[4] J. Erickson, D. Guoy, J. M. Sullivan, and A. ngr, "Building space-time meshes over arbitrary spatial domains," *Engineering with Computers*, vol. 20(4), pp. 342–353, 2005.

[5] R. Abedi, S. Chung, J. Erickson, Y. Fan, M. Garland, D. Guoy, R. Haber, J. Sullivan, S. Thite, and Y. Zhou, "Spacetime meshing with adaptive refinement and coarsening," *Proc. 20th Annual Symposium Computational Geometry*, pp. 300–309, 2004.

[6] S. Thite, "Spacetime meshing for discontinuous galerkin methods," Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, Champaign, May 2005. [Online]. Available: https://arxiv.org/abs/0804.0942

[7] A. Mont, "Adaptive unstructured spacetime meshing for four-dimensional spacetime discontinuous galerkin finite element methods," M.S. thesis, Univ. of Illinois at Urbana-Champaign, Champaign, May 2011. [Online]. Available: https://www.ideals.illinois.edu/bitstream/handle/2142/29810/Mont_Alexander.pdf?sequence=1&isAllowed=y

[8] B. Grnbaum, *Convex Polytopes, Second Edition*, ser. Graduate Texts in Mathematics. Springer, 2003, no. 221.

[9] G. M. Ziegler, *Lectures on Polytopes*, ser. Graduate Texts in Mathematics. Springer, 2007, no. 152.

[10] J. R. Munkres, *Elements Of Algebraic Topology.* Westview Press, 1993, no. 1.

[11] "Simplicial homology." [Online]. Available: https://www.fields.utoronto.ca/programs/scientific/04-05/data_sets/parent.pdf

[12] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry, Algorithms and Applications.* Springer, 2008.

[13] V. Dyedov, N. Ray, D. Einstein, X. Jiao, and T. J. Tautges, "Ahf: array-based half-facet data structure for mixed-dimensional and non-manifold meshes," *Engineering with Computers*, vol. 31(3), pp. 389–404, 2015.

[14] S. Thite, S. Chung, J. Erickson, and R. Haber, "Adaptive spacetime meshing in $2d\times$time for nonlinear and anisotropic media," *8th US National Congress on Computational Mechanics, Minisymposium on Mesh and Geometry Generation*, 2005.

[15] J. Palaniappan, R. B. Haber, and R. L. Jerrard, "A space-time discontinuous galerkin method for scalar conservation laws," *Computer Methods in Applied Mechanics and Engineering*, vol. 193, pp. 3607–3631, 2004.

[16] S. T. Miller and R. B. Haber, "Space-time discontinuous galerkin method for hyperbolic heat conduction," *Computer Methods in Applied Mechanics and Engineering*, vol. 198, pp. 194–209, 2008.

[17] J. Palaniappan, S. T. Miller, and R. B. Haber, "Sub-cell shock capturing and space-time discontinuity tracking for nonlinear conservation laws," *Int. J. Numerical Methods in Fluids*, vol. 57, pp. 1115–1135, 2008.

[18] S. T. Miller, B. Kraczek, R. B. Haber, and D. D. Johnson, "Multi-field spacetime discontinuous galerkin methods for linearized elastodynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, pp. 34–47, 2009.

[19] R. Abedi, R. B. Haber, and B. Petracovici, "A spacetime discontinuous galerkin method for elastodynamics with element-level balance of linear momentum," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, pp. 3247–3273, 2006.

[20] R. Abedi, R. B. Haber, S. Thite, and J. Erickson, "An h-adaptive spacetime-discontinuous galerkin method for linearized elastodynamics," *European Journal of Computational Mechanics*, vol. 15(6), pp. 619–642, 2006.

[21] R. Abedi, S. Chung, M. A. Hawker, J. Palaniappan, and R. B. Haber, "Modeling evolving discontinuities with space-time discontinuous galerkin methods," *IUTAM Symposium on Discretization Methods for Evolving Discontinuities*, vol. 5 of *IUTAM Bookseries*, pp. 59–87, 2007.

[22] R. Abedi, M. A. Hawker, R. B. Haber, and K. Matou, "An adaptive spacetime discontinuous galerkin method for cohesive models of elastodynamic fracture," *Int. J. Numerical Methods in Engineering*, vol. 81, pp. 1207–1241, 2010.

[23] R. Abedi and R. B. Haber, "Spacetime dimensional analysis and self-similar solutions of linear elastodynamics and cohesive dynamic fracture," *Int. J. Solids and Structures*, vol. 48, pp. 2076–2087, 2011.

[24] P. Clarke and R. Abedi, "Modeling the connectivity and intersection of hydraulically loaded cracks with in situ fractures in rock," *International Journal for Numerical and Analytical Methods in Geomechanics*, vol. 42(14), pp. 1592–1623, 2018.

[25] R. Abedi and S. Mudaliar, "An asynchronous spacetime discontinuous galerkin finite element method for time domain electromagnetics," *Journal of Computational Physics*, vol. 351, pp. 121–144, 2017.

[26] R. Pal, R. Abedi, A. Madhuhkar, and R. B. Haber, "Adaptive spacetime discontinuous galerkin method for hyperbolic advection-diffusion with a non-negativity constraint," *International Journal for Numerical Methods in Engineering*, vol. 105(13), pp. 963–989, 2016.

[27] P. Gottschling, *Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers*, ser. C++ In-Depth Series. Addison-Wesley, 2016.

[28] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, ser. C++ In-Depth Series. Addison-Wesley, 2001.

[29] D. D. Gennaro, *Advanced Metaprogramming in Classic C++.* Apress, 2015.

[30] J. R. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational Geometry: Theory and Applications*, vol. 22, issues 1-3, pp. 21–74, May 2002.

[31] H. Si, "Tetgen, a delaunay-based quality tetrahedral mesh generator," *ACM Trans. on Mathematical Software*, vol. 41 (2), Article 11, 36 pages, 2015.

[32] R. Abedi and R. B. Haber, "Spacetime simulation of dynamic fracture with crack closure and frictional sliding," *Advanced Modeling and Simulation in Engineering Sciences*, vol. 5, p. 22, 2018.

[33] D. A. Field, "Laplacian smoothing and delaunay triangulations," *Communications in Applied Numerical Methods*, vol. 4(6), pp. 709–712, 1998.

[34] P. Cignoni, D. Costanza, C. Montani, C. Rocchini, and R. Scopigno, "Simplification of tetrahedral meshes with accurate error evaluation," in *Proceedings Visualization 2000. VIS 2000*, Salt Lake City, UT, USA, Oct. 2000, pp. 85–92.

[35] "Finding holes in data," 2016. [Online]. Available: http://www.ams.org/publicoutreach/feature-column/fc-2016-12

[36] "Stratified spaces," 2016. [Online]. Available: https://ocw.mit.edu/courses/mathematics/18-965-geometry-of-manifolds-fall-2004/lecture-notes/lecture12.pdf

[37] S. Weinberger, *The Topological Classification of Stratified Spaces*, ser. Chicago Lectures in Mathematics. The University of Chicago Press, 1994, no. 1.

[38] C. C. Pinter, *A Book of Set Theory.* Dover Publications, Inc., 2014.

[39] F. H. Croom, *Principles of Topology.* Dover Publications, Inc., 2016.

[40] D. Nash, *A Friendly Introduction to Group Theory.* CreateSpace Independent Publishing Platform, 2016.

[41] D. Braess, *Finite Elements: Theory, fast solvers, and applications in solid mechanics.* Cambridge University Press, 2007.

[42] B. Q. Li, *Discontinuous Finite Elements in Fluid Dynamics and Heat Transfer*, ser. Computational Fluid and Solid Mechanics. Springer, 2006.

[43] T. H. Cormen, C. H. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* MIT Press, 2009.

[44] E. Gossett, *Discrete Mathematics with Proof.* Pearson Education, Inc., 2003.

[45] C. C. Pugh, *Real Mathematical Analysis.* Springer, 2015.

[46] G. E. Shilov, *Linear Algebra.* Dover Publications, Inc., 1977.

[47] B. Stroustrup, *The C++ Programming Language, Fourth Edition (C++11).* Pearson Education, Inc., 2013.

[48] M. Reddy, *API Design for C++.* Elsevier, Inc., 2011.

[49] K. Guntheroth, *Optimized C++: Proven techniques for heightened performance.* O'Reilly Media, Inc., 2013.

[50] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve your use of C++11 and C++14.* O'Reilly Media, Inc., 2015.

[51] V. Sehr and B. Andrist, *C++ High Performance.* Packt Publishing, 2018.

[52] A. Williams, *C++ Concurrency in Action: Practical Multithreading.* Manning Publications Co., 2012.