Imperial College London
Department of Earth Science Engineering
MSc in Applied Computational Science and Engineering

Independent Research Project
Final Report

Conservative Interpolation Between Unstructured Meshes that Preserves Heterogeneity and Structure for Modeling Three-Dimensional Bone

Nourhan Berjawi
nb621@ic.ac.uk

Supervisor:
Dr. Adriana Paluszny
Dr. Robin Thomas

June 2022

Abstract

This project develops a supermeshing algorithm to aid the conservative interpolation between two 3D tetrahedralized meshes. This is inspired by the work of Farell et al., who introduced the supermeshing technique for conservative mapping [1]. Building on their work, I have studied the general requirements of creating a supermesh in 3D. Being able to produce accurate supermeshes for complex grids could speed up 3D simulations, and increase their accuracy, as global integrals across the domain would be conserved. Hence, an algorithm called the "Intersecting Tetrahedra Supermeshing Method" is designed. It uses subroutines that can detect tetrahedron-tetrahedron intersections [11], as well as find the intersection points using standard computational geometry techniques. The proposed supermeshing approach is discussed in detail, and the two subroutines are tested for correctness and speed.

1 Introduction

Discretizing complex 3D geometries into unstructured meshes is a challenging but vital process in computational science. This numerical discretization allows for solving systems of equations that model real-life processes such as weather changes, fluid dynamics, and propagation of fractures in bones. Like every computational problem, solving a large system of equations requires attention to memory consumption, accuracy, and speed. Representing processes that vary each fraction of the second necessitates finding non-traditional ways of solving systems involving large unstructured meshes. Conservative interpolation is a popular approach to accomplish that since it has proven to be less diffusive for discontinuous fields than other methods such as node-wise or Grandy interpolation [14]. This method ensures that the global integrals of the system are conserved, therefore preserving conservation properties under discretization. Farell et al. proposed a conservative interpolation method that uses an auxiliary mesh called a supermesh to map between two grids [1].

This project aims to facilitate interpolation between 3D unstructured meshes representing a section of a bone that has undergone a fracture. When a bone is subjected to a hit (or anything else that might cause it to break), a fracture will gradually grow with time, which evidently changes the mesh that represents this bone. In examining this propagation, we would be working on a very small scale [2]. Consequently, we end up having very large unstructured meshes stored in memory. Given that a small time step is taken in the simulation process to reduce the discretization error, this ends up being a costly task in terms of execution time [3]. To mitigate this issue, we can instead use conservative interpolation using a supermesh [1]. In this method, we begin by constructing a supermesh of both grids representing the current timestep and the next one. No out-of-the-box C++ supermesh generator can do this, and this project focuses on developing such an algorithm.

1.1 Previous Attempts at Creating/Using a 3D Supermesh

Rinaldi et al. presented a flux-conserving method for non-conformal mesh block interfaces [4]. The technique can be used for the finite-volume discretization of any system of conservation laws and uses an auxiliary 3D supermesh. This method does not involve any interpolation and, by construction, guarantees conservation. It works by replacing the parent meshes in the flux balance calculation with the supermesh. In this method, the 3D elements may have either quadrilateral or triangular faces. The supermesh is constructed by intersecting all elements of mesh A and B together. For each pair, the vertices of EA contained in EB are found and vice-versa. Then the edges are intersected, and any duplicate points are removed. Lastly, if the resulting number of elements is greater than 2, new supermesh elements are defined.

Oner et al. present another example of using a supermesh [5]. A parallel adaptive mesh refinement strategy is incorporated into the side-centered finite volume method to obtain highly accurate numerical results for viscoelastic flow problems. This approach for solving viscoelastic problems uses conservative interpolation and employs supermeshes as intermediaries to facilitate the interpolation. To do so, the parent geometries are split into hexahedral volumes and later divided into triangular shapes in the refinement and supermesh. The Delaunay triangulation method is not used here, as it has proved complex for 3D geometries. Instead, element-by-element intersection calculations are done. A template-based side-centered refinement algorithm is proposed. The mesh elements are looped over, and a check is done for whether the parent element requires coarsening. If so, all children get set as inactive. Then for all the elements needing refinement, based on the number of nodes and level of refinement, a template is selected, and the element is refined; this is repeated until there are no more unrefined elements.

Menon et al. describe a method for conservatively remapping fields from one mesh to another for finite-volume computations [6]. The proposed method is also second-order accurate. The base and target meshes are not assumed similar, making this approach general at the high cost of calculating many intersections. The parent meshes are also assumed to be composed of polyhedra, which are then decomposed into tetrahedra. Bad elements are then agglomerated and remeshed in order to avoid computational errors later and reduce the number of intersection calculations. Afterward, mappings are created to match the children tetrahedra to their polyhedral parents.

Evidently, supermeshing has been used for solving many computational problems. Conservative interpolation using a supermesh is a promising candidate for simulating bone fractures since most fractures are modeled at a microscopic level. The trabecular (porous honeycomb-like core) and cortical (stiff external shell) structures necessitate representing a complex geometry when modeling the fractures [7]. At such a small scale, this simulation becomes highly costly in terms of memory and time. In section 2.1, the reason for using tetrahedral supermeshes is explained.

1.2 Theoretical Background for Conservative Interpolation Using a Supermesh

There exists a robust theoretical background for the supermeshing technique, and how it can speed up mesh adaptivity and increase its accuracy [1]. However, the detailed process of supermeshing is explained only for the 2D case. It certainly extends to the 3D case, with some additions and modifications.

A supermesh denoted $\mathcal{T}_c$ of parent meshes $\mathcal{T}_A$ and $\mathcal{T}_B$ is defined as any mesh that fulfills the following 2 conditions:

(1) Any node $\mathcal{N}$ present in the parent mesh must exist in the supermesh. (i)
(2) The intersection of an element of $\mathcal{T}_c$ with any element of either of the 2 parent meshes must either be the empty set or the whole element $\mathcal{K}_c$. (ii)

Assuming such mesh is generated, to go forth with conservative interpolation, it is also essential to have mappings that can take any element in the supermesh and find its parent elements in the base and target meshes. Thus, the complete algorithm consists of three significant steps. We start by creating a supermesh $\mathcal{T}$c as well as the mappings $\mathcal{X}_{CA}$ and $\mathcal{X}_{BC}$. Then, for each element in $\mathcal{T}_B$, we find its integral value as the sum of that of its children in $\mathcal{T}$c. Finally, after

finding the elemental integrals, the nodal values are computed through a Galerkin Projection, explained in detail by Farell et al. [1].

We are considering mesh $\mathcal{T}_A$ to be the base mesh and $\mathcal{T}_B$ to be the target mesh; the goal is to interpolate the values from $\mathcal{T}_A$ onto $\mathcal{T}_B$. Given a function $q(x)$, the integral of which is to be conserved in the discretized process of interpolation, the sum of the elemental integrals over $\mathcal{T}_A$ must be equal to that over $\mathcal{T}_B$, under a projection π_B.

$$\sum_{K_A \in \mathcal{T}_A} \int_{K_A} q(\boldsymbol{x})\ dV = \sum_{K_B \subseteq \mathcal{T}_B} \int_{K_B} \Pi_B[q](\boldsymbol{x})\ dV$$

Since each element in $\mathcal{T}_B$ can be expressed as the union of its children in $\mathcal{T}_C$, we can therefore replace each elemental integral of $\mathcal{T}_B$ with the sum of the integrals of its children in $\mathcal{T}_C$, with π_C being the projection operator applied to the supermesh. Since no two child elements in $\mathcal{T}_C$ intersect, the problem boils down to interpolating from $\mathcal{T}_A$ to $\mathcal{T}_C$.

$$\sum_{K_A \in \mathcal{T}_A} \int_{K_A} q(\boldsymbol{x})\ dV = \sum_{K_C \in \mathcal{T}_C} \int_{K_C} \Pi_C[q](\boldsymbol{x})\ dV$$

Thus, for this process to be conservative, all that is left to do is find an operator that preserves the integral over the meshes from mesh A to mesh C. One such operator is defined as the fraction of the integral of the child element in C over that of its parent in A. Assuming the function $q(x)$ is constant over a single element, the nodal values across $\mathcal{T}_B$ can then be recovered, and for each node, the value would be the integral across the element divided by the element's volume. Afterward, a Galerkin projection can be applied to obtain piecewise linear basis functions from constant elemental values.

2 Methodology

2.1 Process of Creating a Valid Supermesh

When working in 2D, given two meshes $\mathcal{T}_A$ and $\mathcal{T}_B$, for their supermesh $\mathcal{T}_C$ to be valid, all the nodes and edges of the parent meshes must be present in $\mathcal{T}_C$. This is proven in lemma 6 by Farell et al [1]. This lemma leads to the following algorithm.

The general steps for constructing the supermesh in 2D are as follows:

(1) Obtain the union $\mathcal{N}$ of all the nodes of $\mathcal{T}_A$ and $\mathcal{T}_B$
(2) Obtain the union $\mathcal{D}$ of all the faces of $\mathcal{T}_A$ and $\mathcal{T}_B$
(3) $\mathcal{N}$ and $\mathcal{D}$ form an imperfect piece-wise linear complex
(4) $\mathcal{N}$ and $\mathcal{D}$ are given to the Delaunay Triangulation method of the Triangle Library, and a triangulated supermesh is obtained

An underlying step in this algorithm is creating the parenthood mappings for the resulting supermesh's elements. Since each edge can be shared by a maximum of 2 elements, the edges are annotated with an encoding of two integers representing the parent elements in the original meshes. This encoding is a bijection from which the associated input elements are easily recoverable [8].

When working in 3D, there are slight modifications to be done. From lemma 6 mentioned above, one can infer that in 3D, a valid supermesh must have all the nodes of the parents, and all the faces, not only edges. This is easily achievable for any 2 arbitrary 3D meshes, by simply finding the union of their nodes and the union of their triangular faces. However, the Triangle library used by Farell et al. only works for 2D geometries. In this work, TetGen is used to generate

tetrahedralized parent meshes in 3D. Nevertheless, it does not offer the option of tetrahedralizing a certain geometry, whilst keeping certain faces. Therefore, a supermeshing algorithm is constructed from scratch.

2.2 Method of Intersecting Tetrahedra

2.2.1 Motivation for Algorithm

A common technique for tetrahedralizing a domain is the Delaunay Method. The method takes a set of points $\mathcal{S}$ in space and returns a list of tetrahedra such that each simplex (edge, triangle, or tetrahedron) follows this rule: all its vertices can be connected by a circumsphere that includes no other vertex [9]. As a result, this algorithm produces mostly “round” tetrahedra, and not so many “thin” ones [10]. The C++ library TetGen has an interface for using this algorithm. One can simply provide the vertices that form the exterior of the domain, or the Piecewise Linear Complex, and TetGen then decomposes this domain into Delaunay tetrahedra. One can also provide all the nodes (interior and exterior) of a mesh and get a tetrahedralization. TetGen also provides options for refining an existing mesh. One downside, however, is that TetGen does not allow us to fix certain faces of the domain.

One approach to producing a supermesh using TetGen would be to get the union of all nodes of both parent meshes and pass them on to TetGen’s tetrahedralize() function. This approach of course does not work because, as mentioned earlier, the presence of all the faces of the parents in the supermesh is essential for its validity.

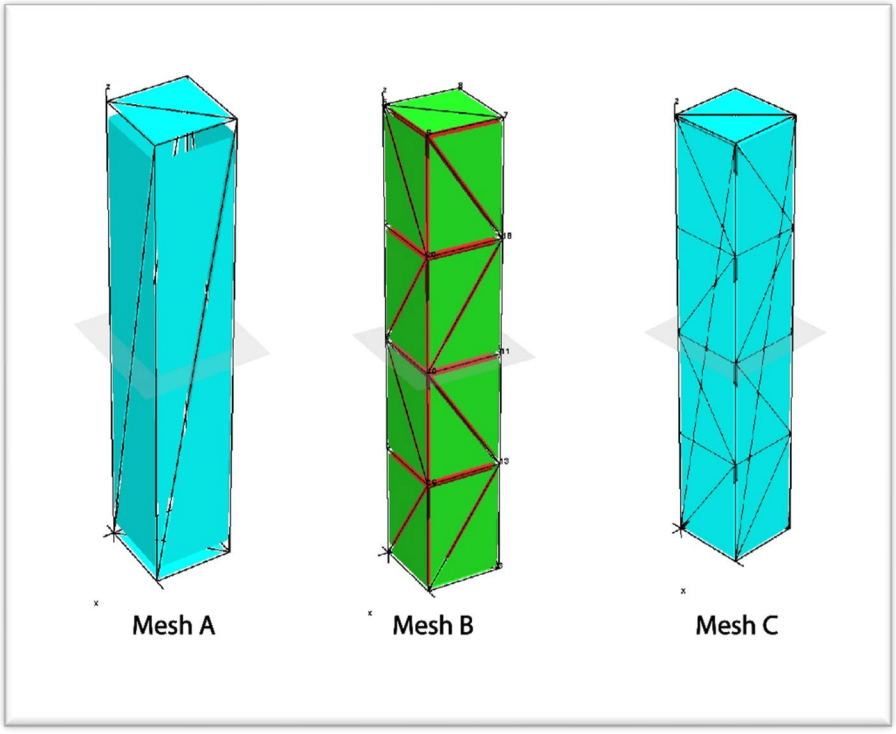

Figure 1 In this figure, Mesh A and Mesh B are two domains that were tetrahedralized by TetGen, with Mesh B being a more refined mesh. Mesh C shows the union of all the nodes of A and B as well as the union of their faces.

The method discussed in this project involves getting two parent meshes that have already been tetrahedralized by TetGen, and then manually finding the tetrahedralization of their union.

The resulting children tetrahedra at each intersection are repeatedly intersected with other parts of the domain until there are no more intersections. Each tetrahedron in the supermesh has a member called parents, encoding its parent elements in meshes A and B.

2.2.2 Description of Algorithm

Despite being developed with input meshes being a result of TetGen's tetrahedralization method, the intersecting tetrahedra algorithm makes no assumptions about the input meshes in terms of the shapes of the domain. It does not require the input to be a Delaunay Tetrahedralization specifically and has no constraints on the number of elements of each parent mesh.

The algorithm works in a recursive manner and makes use of three meshes (or lists of tetrahedra), the first two lists representing the elements of $\mathcal{T}_A$ and $\mathcal{T}_B$, and the third one being an auxiliary list of tetrahedra that starts out empty, and gets progressively filled with elements as the algorithm progresses. The nodes of both parents as well as their elements (which are essentially groups of 4 nodes representing each original tetrahedra in the parent mesh) are first received. Then for each element of mesh $\mathcal{T}_A$, two major steps are done. Firstly, the element $\mathcal{K}_i$ of $\mathcal{T}_A$ is intersected with the auxiliary mesh, and child tetrahedra are obtained. The child tetrahedra are then intersected with the elements of $\mathcal{T}_B$.

2.2.3 Pseudocode of Supermesh Algorithm and Notes

Algorithm 1 Intersecting Tetrahedra Algorithm for Generating a Supermesh

```
function supermesh(A, B):
    C = [ ] empty vector of tetrahedra
    marker = array of bools of size sizeof(B)
    for i in range(sizeof(A))
        AC , AB = [ ]  empty vectors of int
        temp = [ ] empty array of tetrahedra
        t , c = 0

        // start of BLOCK I
        if (AC ! empty)
            for j in range(sizeof(C))
                if A(i) intersects C(j)                *
                    temp.push(C(j))
        for j in range(sizeof(B))
            if (!marker[j])
                if A(i) intersects B(j)                *
                    AB.push(j)

        // start of BLOCK II
        while true
            if (temp[t] intersects C[AC[c]])           *
                X = get_intersection (temp(t),C[AC[c]] )
                if sizeof(X) == 1, go to else
                temp.remove(t)
                temp.push(X)
            else
                c++
                if ( c == sizeof ( AC ) )
                    c = 0;  t++
                    if ( t == sizeof ( temp ) )
                         break

        // start of BLOCK III
        temp1 = [ ] empty vector of tetrahedra


        for j in range ( sizeof ( temp ) )
            if (one of temp(j)'s parents is unintialized)
                    temp.remove(temp(j))
                    temp1.push(temp(j))
        t , c = 0
        if ( AB ! empty )
            while true
                if ( temp1(t) intersects B(AB[c]) )  *
                    X = intersection temp1(t) and B(AB[c])
                    if sizeof(X) == 1, go to else
                    temp1.remove(t)
                    temp.push(X)
                else
                    c ++
                    if ( c == sizeof(AB))
                        c = 0
                        t++
                        if ( t == sizeof (temp1))
                           break
        // start of BLOCK IIV
        for j in AB
           marker[j] = 1
        C.remove(all elements pointed at by AC and AB)
        C.push(temp, temp1)
```

In Algorithm 1, the two parent meshes A and B are taken as input, and an empty auxiliary mesh C is initialized. No assumptions are made about the order of the elements of the parent meshes, their orientation, or the order their nodes are stored in. The only assumptions made are that no

two elements of A or two elements of B intersect, and that no elements in A or B are degenerate tetrahedra (have 4 coplanar points, or 3 or more collinear points).

The algorithm can be split into 4 blocks: initialize (block I), intersect with C (block II), intersect with B (block III), and update C (block IV). All four of these blocks are repeated for each element of mesh A.

Block I determines which elements of C and B does element i of A intersect. The ranks of these elements are saved in two separate arrays. An empty temporary vector of elements is initialized (temp can be thought of as a queue linked list, but for ease of implementation, an STL vector can be used).

Block II loops over each element of temp and checks if it intersects the predetermined elements of C. Each time an intersection occurs, the intersection function returns one or more tetrahedra, each with either two parents assigned, or only one, the one parent being $\mathcal{K}_i$ of A. Then, the intersecting element is removed from temp and the resulting child elements are added to temp. The process keeps repeating until all of temp is checked for intersection with the predetermined elements of C.

Block III is similar to block 2. It takes the children elements that resulted from intersecting $\mathcal{K}_i$ of A with C, such that these elements have been assigned only one parent. Then, the intersection of these child elements with the predetermined elements of B is done in the same way as before.

Block IV updates the elements of the supermesh, and marks the intersected elements of B to signify they are now part of the supermesh; no need to check them as part of mesh B.

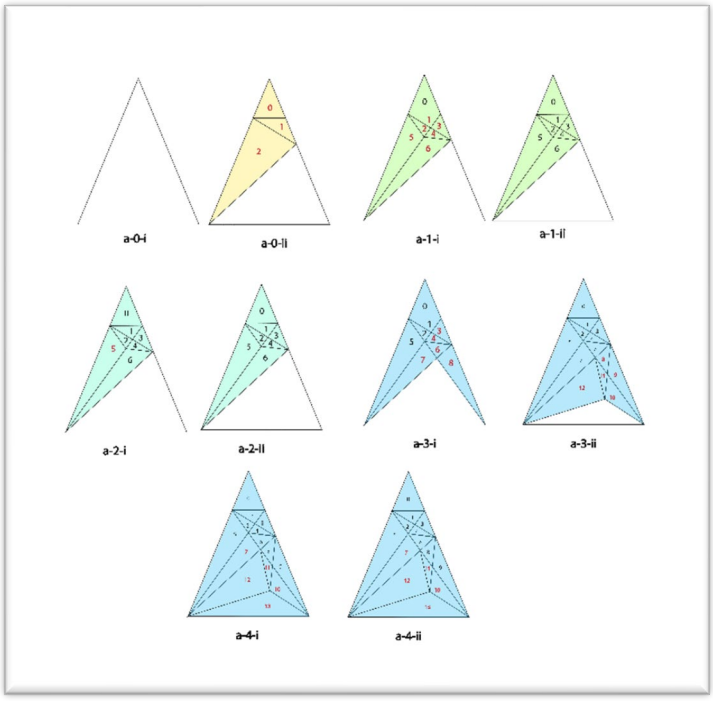

Figure 2 Shows an example of how the proposed supermeshing technique works. 2D meshes are used for demonstrative purposes.

2.2.4 Subroutines of Supermesh Algorithm

In the supermeshing process, three essential functions do most of the work. The first function is the boolean intersection function. This function takes two tetrahedra and checks whether they intersect. This function is used in the lines marked with (*) and is called repeatedly for many elements each iteration. The way to check if two tetrahedra intersect is to essentially check if each edge of B intersects any face of A and vice versa. Checking whether an edge intersects a face repeatedly is an expensive process. A faster way is proposed by Ganovelli et al. [11]; this method makes use of the separating axis theorem but avoids some computations that are usually done while performing the separating axis test.

The separating axis theorem is used to detect the overlap of convex polytypes. It states that if the convex polyhedra are separable, there exists an axis on which their projections do not overlap. This axis is either orthogonal to the face of one of the two convex bodies or to an edge for each of the two. [11] These two rules are theoretically certainly a step up from having to do edge-face intersections. However, computationally, determining if a separating axis orthogonal to a face or orthogonal to a pair of edges exist is costly. The algorithm proposed by Ganovelli et al. is called GPR; its key advantage is that it does not explicitly test pairs of edges to find a separating axis. The algorithm instead checks if each pair of two faces of tetrahedra is a separating plane, saves some results, then in no separating axis is found, the results are used to determine if the edge shared by these two faces is separating.

The GPR algorithm uses two main functions, a face(i) method that determines whether the plane the ith face of tetrahedron A lies on is a separating plane. This is done by first finding the normal to the face pointing outside the tetrahedron, and then for each point of tetrahedron B, the dot product is used to determine on which side of the plane the point lies. If all points are on the outer side of any face, then the plane is separating, and the tetrahedra do not overlap. The position of each point of B with respect to the ith face of A is saved in a bitmap called a mask, and the dot products are also saved. In graph 2 of figure 2, we can see how looking at the domain from the point of view parallel to the plane of ABC; we can see how a dot product with the normal can tell us the location of the points of B. The second important method is edge(i,j) which tests if the edge shared by the ith and jth faces of tetrahedron A is contained within a separating plane. In graph 3 of figure 2, we see how this can be determined by splitting the domain into four parts. Edge(i,j) is called after the calls to face(i) and face(j) have been completed. Thus, the results are reused and combined with bitwise operations and comparisons to find whether a separating plane containing the shared edge exists. For more information about this algorithm, please refer to Ganovelli et al. [11]

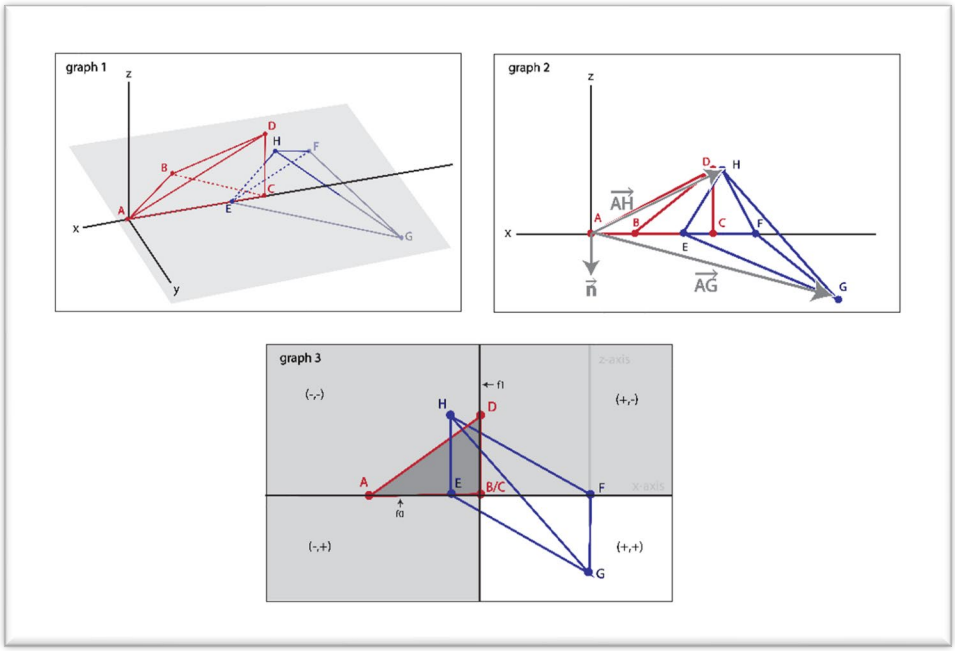

Figure 3 Demonstration of the how Ganovelli et al.'s GPR algorithm works; graph 1 shows the two tetrahedra A and B in red and blue respectively; graph 2 shows triangle ABC of tetrahedron A as a potential separating axis; graph 3 shows the domain split into 4 parts by segment B/C

The second important function in the supermeshing process is get_intersection_points(A, B) which takes two tetrahedra as input and returns a list of their intersection points. This function intersects every edge of B with every face of A and vice versa. All in all, 48 edge-face pairs are checked. This function includes a lot of computational geometry subroutines: For each edge-face pair, specific tetrahedral volumes are calculated using tet_vol() to assess whether the edge and segment are coplanar, meaning this becomes a 2D problem and, if they are not, whether they intersect in 3D. If they are coplanar, each side of the triangle is checked for an intersection with the edge using a segment_segment() intersection function that returns a boolean value. If there is a side-edge intersection, we intersect the two lines formed by the two segments using get_line_line(), which checks if any extremities of segment 1 lie on segment two and vice versa using point_on_line(). If point_on_line() doesn't return a point for any case, the intersection point is calculated using the equations of the two lines. The resulting point to the vector of points to be returned. If the edge-face pair are not coplanar, we use the signs of the volumes calculated earlier to decide if they intersect. If they do, we call line_plane(), get the intersection point, and add it to our result.

Figure 4 shows the function calls done in get_intersection_points(). The bulk of the computation takes place in the five functions mentioned at the bottom of the figure.

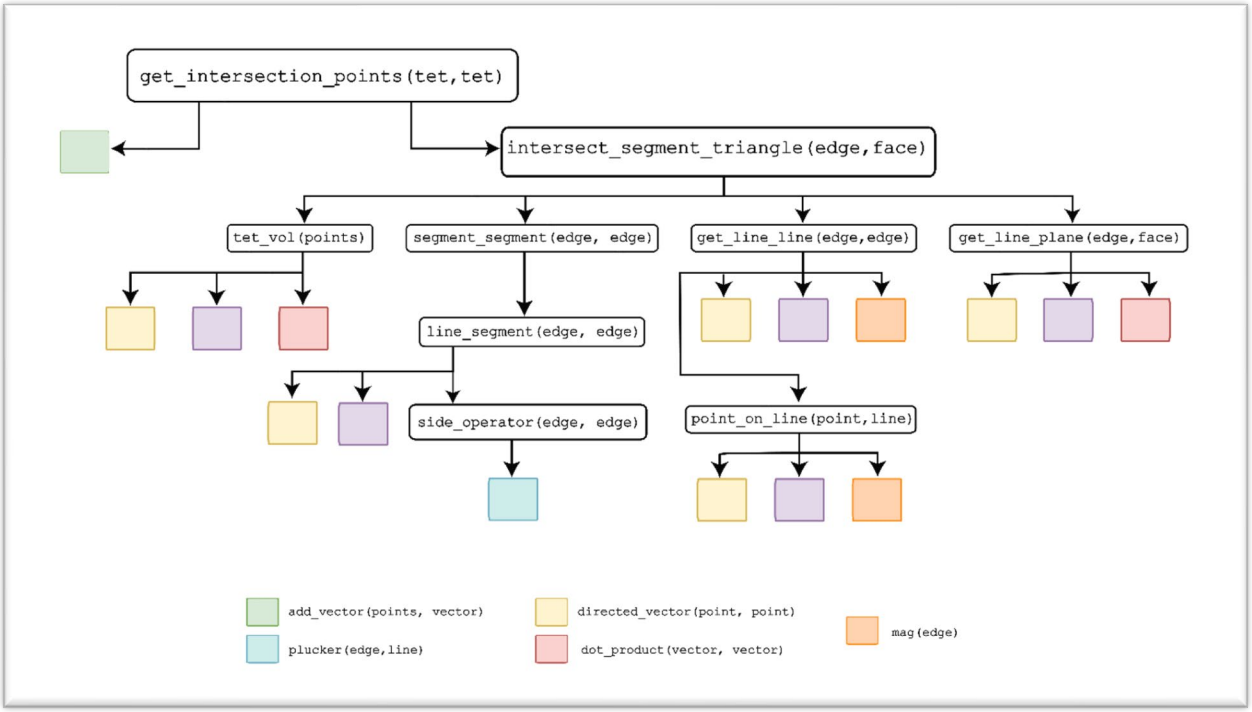

Figure 4 shows the function calls done in the get_intersection_points() method.

The third important function is one that takes 2 tetrahedra and their intersection points, and based on the finite number of ways 2 tetrahedra can intersect, returns a list of children tetrahedra. This was inspired by the work of McCoid et al. [13] in which they prove that only 0, 3, or 4 intersections may exist between all of the edges of a tetrahedron X and a plane of another tetrahedron Y. With some additional constraints, this can lead to a general breakdown of how any two tetrahedra can intersect. Its code has not yet been implemented.

2.3 Library Code Breakdown

2.3.1 Code Meta Data

The code can be found here: https://github.com/ese-msc-2021/irp-nb621.git

It was developed with and tested on WSL 1 shell on Windows 10, with Ubuntu 20.04.4 LTS. The CPU on the HP machine used has 4 cores and 8 logical processors, with a base speed of 2.80 GHz.

The code is wholly in C++, and the compiler used is GNU compiler GCC version 9.4.0.

To compile the tests found in irp-nb621/code/library/tests, run the following command two commands:

```
g++ ../source/node.cpp ../source/face.cpp ../source/ele.cpp ../source/supermesh.cpp
test_random_tetrahedra.cpp -o test
```

```
./test
```

The code is documented sufficiently in the header files in irp-nb621/code/library/source.

For compiling the file bar_mer/driver.cpp, please check irp-nb621/documents/tetgen_manual.pdf page 23 and compile TetGen in the same directory. Then run:

```
g++ ../source/node.cpp ../source/face.cpp ../source/ele.cpp -o test tetcall.cxx -L./ -ltet

./test
```

2.3.2 Structure of Library

All the methods mentioned in this paper can be found in code/library/source. This directory includes five header files that represent nodes, faces, elements, tetrahedra, and supermeshes. Each header file contains a class of the same name.

It is assumed that the user will use TetGen to generate the two tetrahedralized parent meshes. Afterward, the resulting tetrahedra are saved to .node, .ele, and .face files by TetGen. The node.cpp and ele.cpp files have functions can read each of these types of files and return an array of nodes or eles. The supermesh routine does not use the faces class. This class was created at the beginning of the project in order to be able to see how combining the faces of 2 arbitrary meshes' faces would look like; this is shown in figure 1.

There are correctness tests for the 2 large functions intersect() and get_intersection_points(). These tests can be found in code/library/tests. The tests for each function are in a .cpp file of the same name. In that same directory, there is a third file called test_random_tetrahedra.cpp which was used to gain insight about the performance of the two functions mentioned above.

2.3.3 Results

In order to test the intersect() and get_intersection_points() functions, three lists of 2000 random tetrahedra each were constructed. Each list was confined to a cube shaped region of the 3D domain. The second domain intersects the first and third, but the first and third are separable. The tetrahedra generated have no restrictions on the order of their nodes and consequently their faces and edges. I intersected each tetrahedron in list 1 against list 2, and each one in list 2 against list 3, which gave me two equations, with two unknows: time taken by one intersection with no overlap and time taken with overlap.

The results are shown in figure 5. Intersect() functions is faster at detecting no intersection, which makes it useful when deciding whether or not an intersection exists at all. Intersect() is slower for detecting an overlap, because it has to go through all of the rejection tests to prove they intersect. However, since much more tetrahedra do not intersect than the ones that do, it still is useful to use intersect() first as a check.

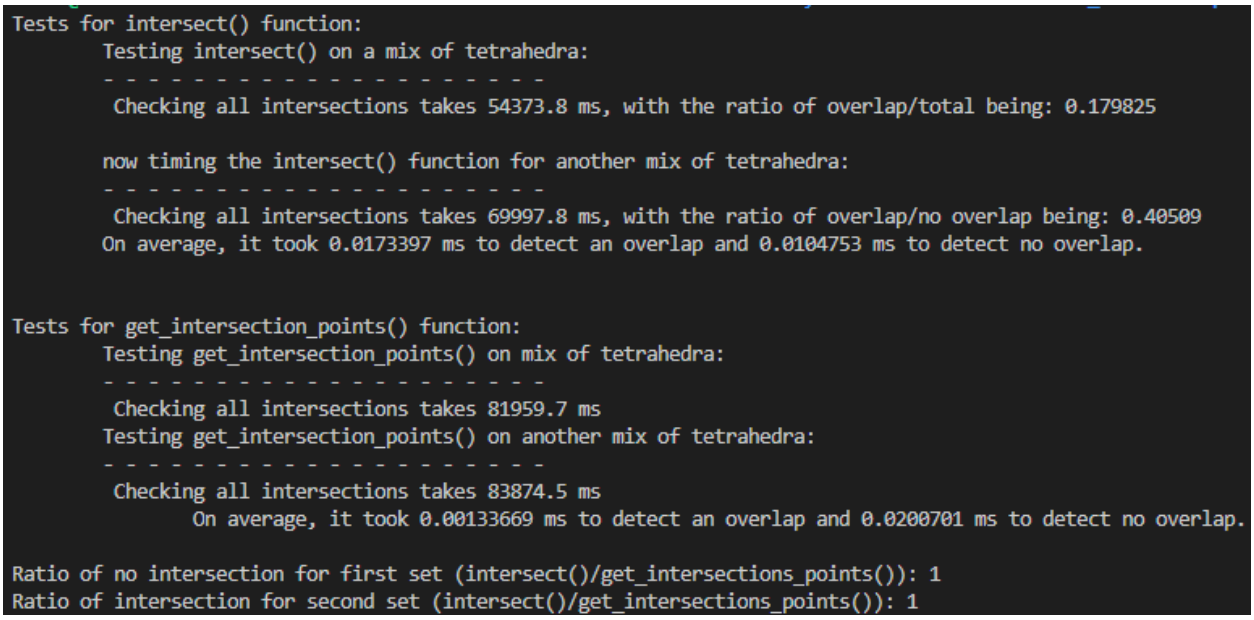

Figure 5 The results of using intersect() and get_intersection_points() on two sets of lists of tetrahedra.

2.3.4 Future Work

The get_intersections_method() is a highly parallelizable process, for it performs 48 asynchronous edge-triangle intersections, which can be done on a thread each. The intersect() method, however, is not parallelizable since the results of each step depend on those of the step before it. The supermeshing algorithm is not parallelizable as well for the same reason; each step changes the supermesh, and those changes are needed for the following step to be correct.

The construct_intersections() algorithm is not developed yet, which is why the full code for the supermeshing method is not developed in the repository. It would certainly be interesting to complete the construct_intersections() routine, as it is specific to tetrahedra which means it can be used side by side with TetGen to create supermeshes, or just simply join two tetrahedra, such that the main faces of the parents are conserved.

3 Conclusion and Discussion

Having a robust and accurate 3D meshing library that is compatible with TetGen, but not dependent on it was one of the goals of the project. There is not much still to be done for the library to be complete. Supermeshing is a theoretically strong, and computationally justifiable technique with promising use for simulations and mesh adaptivity. Hopefully this work paves the way to a complete and well-rounded C++ 3D meshing solution.

The journey from understanding the workings of supermeshing in the context of conservative interpolation to coming up with a viable supermeshing algorithm was lengthy but interesting. Finding a reliable 3D meshing library took a significant amount of time since almost all opensource ones either haven't been updated in a long time or are incredibly tedious to work with. Having found TetGen, there was still a long way ahead filled with learning computational geometry techniques for 3D, which in theory is an easy topic, but in practice has far too many challenges and subtleties. For the past three months during which I researched, implemented,

and assessed my code, I have grown interested in computational geometry and hope to continue learning more in that field in the future.

Bibliography

[1] Farrell, P., Piggott, M., Pain, C., Gorman, G. and Wilson, C. 2009. Conservative Interpolation Between Unstructured Meshes via Supermesh Construction. Computer methods in applied mechanics and engineering 198.33: 2632–2642.

[2] Sabet, F., Raeisi Najafi, A., Hamed, E. and Jasiuk, I. Modelling of Bone Fracture and Strength at Different Length Scales: A Review. Interface Focus, vol. 6, no. 1, 2016, p. 20150055

[3] Li, Gang. Introduction to the Finite Element Method and Implementation with MATLAB®. Cambridge University Press, 2020.

[4] Rinaldi, E., Colonna, P. and Pecnik, R., 2015. Flux-conserving treatment of non-conformal interfaces for finite-volume discretization of conservation laws. Computers & Fluids, 120, pp.126-139.

[5] Oner, E. and Sahin, M., 2016. A parallel adaptive viscoelastic flow solver with template based dynamic mesh refinement. Journal of Non-Newtonian Fluid Mechanics, 234, pp.36-50.

[6] Menon, S. and Schmidt, D., 2011. Conservative interpolation on unstructured polyhedral meshes: An extension of the supermesh approach to cell-centered finite-volume variables. Computer Methods in Applied Mechanics and Engineering, 200(41-44), pp.2797-2804.

[7] Do, X., Hambli, R. and Ganghoffer, J., 2021. Mesh-independent damage model for trabecular bone fracture simulation and experimental validation. International Journal for Numerical Methods in Biomedical Engineering, 37(6).

[8] D.M. Bradley, 2005. Counting the positive rationals: A brief survey. URL <http://www.citebase.org/abstract?id=oai:arXiv.org:math/0509025>.

[9] Verbree, Edward 2010. Delaunay Tetrahedralizations: Honor Degenerate Cases. International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XXXVIII-4/W15

[10] Shewuck, J. Constrained Delaunay Tetrahedralizations and Provably Good Boundary Recovery.

[11] Ganovelli, F., Ponchio, F. and Rocchini, C., 2002. Fast Tetrahedron-Tetrahedron Overlap Algorithm. Journal of Graphics Tools, 7(2), pp.17-25.

[12] Jiménez, J., Segura, R. and Feito, F., 2010. A robust segment/triangle intersection algorithm for interference tests. Efficiency study. Computational Geometry, 43(5), pp.474-492.

[13] McCoid, C. and Gander, M., 2022. Intersection of tetrahedra. [online] Unige.ch. Available at: <https://www.unige.ch/~mccoid/ongoing/Intersection_of_tetrahedra.pdf>

[14] Adam, A., Pavlidis, D., Percival, J., Salinas, P., Xie, Z., Fang, F., Pain, C., Muggeridge, A. and Jackson, M., 2016. Higher-order conservative interpolation between control-volume meshes: Application to advection and multiphase flow problems with dynamic mesh adaptivity. Journal of Computational Physics, 321, pp.512-531.