UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

GENERADOR DE MALLAS DE POLIEDROS EN TRES DIMENSIONES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

ANDRÉS EDUARDO CERDA PIMENTEL

PROFESORA GUÍA :
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN :
MAURICIO PALMA LIZANA
GONZALO NAVARRO BADINO

SANTIAGO DE CHILE
2022

Resumen

Las mallas de tetraedros actualmente son el estándar, tanto para diseños 3D como para simulaciones físicas. Existen bastantes herramientas disponibles para crear, modificar y utilizar en diversas aplicaciones las mallas de tetraedros, no así con las mallas de poliedros. No obstante, la creación de nuevos métodos numéricos (como el método de elementos virtuales), abre la necesidad de tener herramientas que faciliten la creación de mallas de poliedros. Debido a la cantidad reducida de elementos que se utilizan al realizar simulaciones, disminuye el tiempo requerido respecto a mallas convencionales.

Bajo este contexto, se presenta la oportunidad de crear un mallador de poliedros, pero como se dijo anteriormente, existen muchas mallas de tetraedros ya creadas, por lo cual se ve la posibilidad de crear un algoritmo que traspase una malla de tetraedros a una malla de poliedros. En particular se eligió un mallador que produce mallas de buena calidad con el método de tetraedralización de Delaunay, este es el caso de **TetGen**.

Primeramente se parte por tratar de extender un algoritmo para vacíos cosmológicos que utiliza poliedros para su estructura interna, el cual no resultó ser conveniente puesto que al estar orientado a la detección de vacíos, los criterios que utiliza para generar poliedros no llevan a mallas de calidad. Luego se intentó adaptar una versión anterior del mismo, el cual si tiene criterios de generación que sirven para generar poliedros a partir de una malla de tetraedros arbitraria, pero al poseer varias fallas de implementación, lo hizo muy difícil de adaptar.

Es por esto que finalmente se adoptó la decisión de modificar una implementación de otro mallador en python, el cual es similar en implementación pero no en principio al algoritmo que se obtuvo en esta memoria, se le hizo una reingeniería, generando nuevas estructuras de datos, implementando un algoritmo de generación de mallas de poliedros a partir de mallas de tetraedros construidas por TetGen, y exportando dichas mallas a un formato de lectura de poliedros.

Este algoritmo funciona de buena manera, pero requiere más refinamiento para generar mallas de poliedros sin poliedros no simples. El costo de ejecución computacional del algoritmo es $O(n^2)$ según el número de puntos que contenga la malla de input, el algoritmo es inherentemente single-threaded. De todas maneras, el algoritmo es tanto fácil de utilizar como de entender, y su implementación funciona en todos los sistemas operativos actuales gracias a que está hecha en Python.

Agradecimientos

A mis padres por darme las herramientas para enfrentar esta y muchas otras situaciones.

A Carolina por apoyarme, acompañarme y ayudarme.

A mis amigos del DCC, sin los cuales no habría llegado a esta instancia.

A mis amigos de plan común, quienes sufrimos y reímos juntos.

A mis amigos de angels, quienes me han apoyado siempre.

A mis amigos del Calazanz, sin los cuales ni siquiera hubiese entrado a la Universidad.

A mi hermano, por apoyarme en buenas y malas.

A mis hermanas, que me han impulsado a hacer cosas que no me hubiese atrevido.

Al resto de mi familia, algunos que ya no están acá, pero siempre estarán dentro de mí.

Tabla de contenido

Índice de Algoritmos

Índice de ilustraciones

1. Introducción

1.1. Motivación

Las *mallas* son representaciones de un dominio geométrico mediante celdas discretas más pequeñas; son usualmente utilizadas para computar soluciones a ecuaciones diferenciales parciales y renderizar gráficas computacionales, entre otros usos. Un ejemplo de malla es una malla de tetraedros (Ver Figura 1). Un tetraedro tiene 4 vértices, 6 arcos (arcos y aristas se utilizarán de manera intercambiable) y está conformado por 4 caras triangulares. En la mayoría de los casos este tipo de mallas se puede generar automáticamente. Dichas mallas son útiles pues particionan un espacio en distintos elementos sobre los cuales una ecuación diferencial puede ser resuelta mediante soluciones locales que luego aproximan la modelación del fenómeno en el dominio completo.

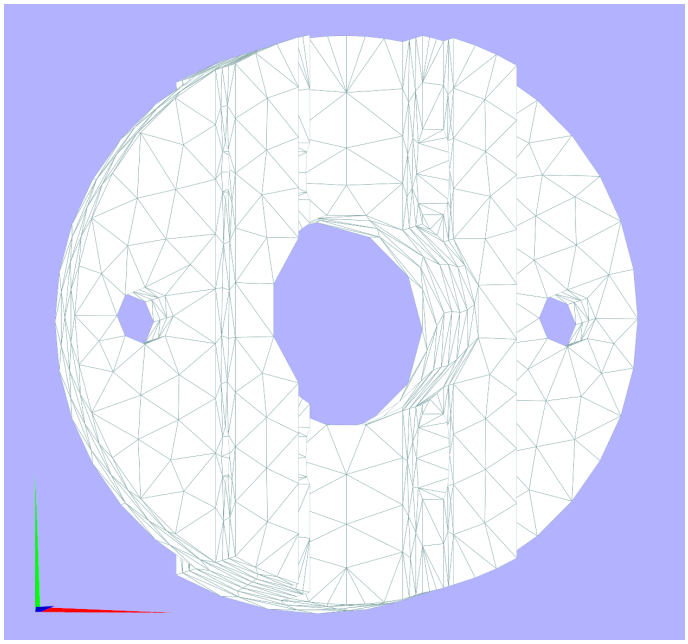


Figura 1: Malla de Tetraedros a partir de un dominio poliédrico en 3D
Renderizado con **Camarón** [4]

El problema es que para simulaciones complejas, las mallas de tetraedros de calidad aceptable pueden generar matrices muy grandes, las cuales toman mucho tiempo para realizar una simulación, por eso se buscan formas de hacer que estas sean más rápidas. Esto se puede hacer bajando la cantidad de puntos a simular (y por ende bajando la precisión de la simulación), o bajando la cantidad de elementos sobre los cuales simular, manteniendo la precisión de la simulación. Esto último se puede hacer reemplazando la malla de tetraedros por una de poliedros, ya que disminuye la cantidad de elementos sobre los cuales se deben ejecutar cálculos.

En esta memoria se propone realizar un algoritmo que reciba como input una malla de tetraedros y la convierta en una malla de poliedros, apta para realizar simulaciones sobre ella, con métodos numéricos tales como el método de elementos virtuales [10].

1.2. Objetivo General

Desarrollar una herramienta que permita generar a partir de mallas de tetraedros mallas de poliedros, las cuales sirvan para ser utilizadas junto al método de elementos virtuales en simulaciones físicas.

1.3. Objetivos Específicos

1. Diseñar e implementar criterios para agrupar conjuntos de tetraedros que definan cada poliedro.
2. Construir la malla de poliedros a partir de los criterios definidos.
3. Estudiar las propiedades de los poliedros generados (si son simples o no).
4. Almacenar las mallas de poliedros generadas en el formato visf para facilitar su visualización.
5. Evaluar las mallas generadas en sus propiedades geométricas.

1.4. Evaluación

Estas mallas son un método relativamente nuevo de simulación, por lo tanto solo existen métricas comparativas con otras mallas en cuanto al número de elementos totales.

Para algunos problemas puede ser apropiado contar con poliedros lo más cercano a *convexo* posible. Esto puede ser interesante además para los usuarios de las mallas si es que llegaran a necesitar englobar ciertos puntos en una cerradura convexa, aunque en general los poliedros generados no tienen porqué ser convexos, pero si simples.

Se incluirán métricas de rendimiento sobre la cantidad de puntos iniciales de la malla, además de métricas que midan el número de poliedros no simples que puedan resultar de un conjunto de puntos aleatorio.

Se evaluará en cuánto se reduce el número de arcos, caras y elementos (tetraedros y poliedros respectivamente) en la malla de poliedros en comparación a la malla de tetraedros inicial.

1.5. Descripción de la solución

La solución consiste en un programa hecho en *Python* inspirado tanto en *DELFIN* [1] como en *POLYLLA* [7]. En una primera instancia se consideró la modificación directa de *DELFIN*, pero como se verá más adelante, la solución final se implementó en python por complicaciones con el código fuente en C++. Esta solución requiere de un procesamiento previo de la malla con *TetGen*, aparte de eso, la solución solo ocupa librerías estándar de *Python*.

1.6. Estructura de la memoria

Los capítulos que siguen tienen la siguiente estructura:

En el capítulo 2 se explican los conceptos necesarios para comprender esta memoria, junto con la explicación de un algoritmo que se referenciará después. En el capítulo 3 se detalla el estado del arte junto con algoritmos que generan distintas estructuras relevantes para esta memoria.

En el capítulo 4 se describe el análisis y diseño de la solución, junto con los requerimientos y la funcionalidad que se espera de ella. En el capítulo 5 se ve la implementación de la solución y sus limitaciones actuales.

En el capítulo 6 se muestran métricas de validación de la solución, junto con gráficos que explican su rendimiento tanto geométrico como en recursos computacionales. Finalmente, en el capítulo 7 se analizan las conclusiones de esta memoria y el trabajo futuro que queda al finalizarla.

2. Conceptos Previos

Algunos conceptos útiles para la comprensión de este informe se listan a continuación:

2.1. Triangulación de Delaunay

Es una malla de triángulos que cumple la condición de Delaunay. Esta condición dice que la circunferencia circunscrita de cada triángulo de la red no debe contener ningún vértice de otro triángulo. Esto llevado a 3D equivale a que la circunesfera de cada tetraedro no contenga ningún vértice de otro tetraedro.

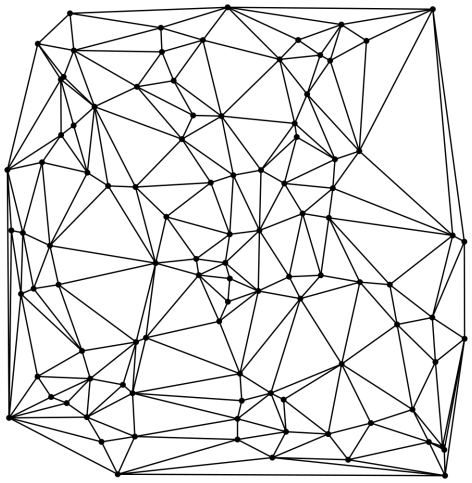
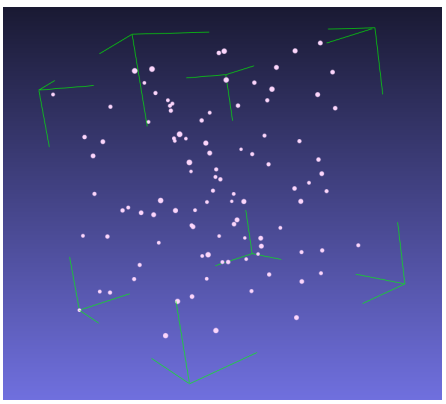


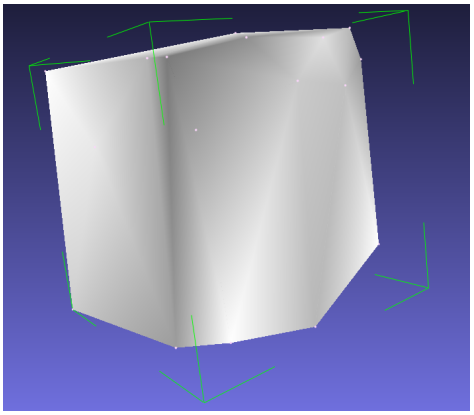
Figura 2: Triangulación de Delaunay de un conjunto aleatorio de puntos

2.2. Envoltura Convexa

Se define como la intersección de todos los conjuntos convexos que contienen a un conjunto de puntos **X**. Es decir, es un espacio convexo, que contiene dentro todo el conjunto de puntos **X**, y a su vez es el menor espacio convexo que contiene al conjunto de puntos **X**.



(a)

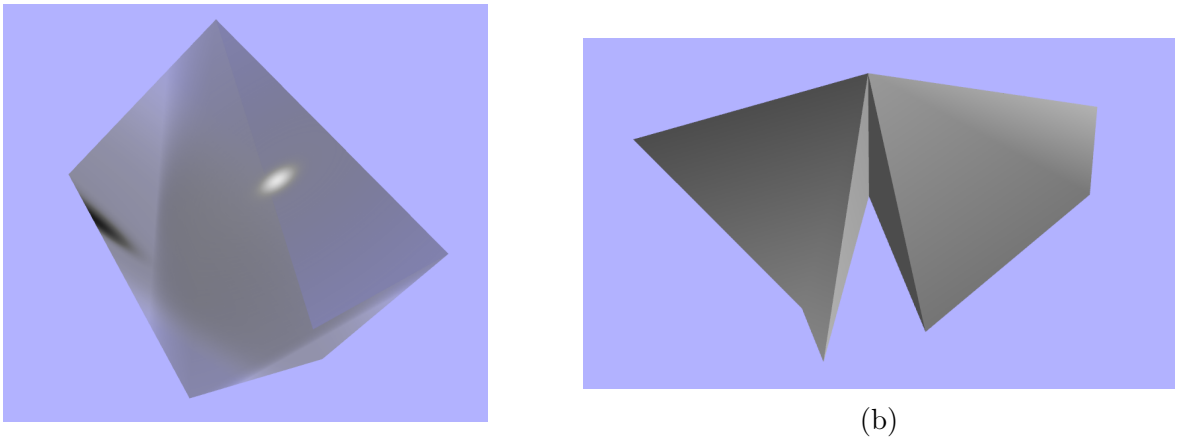


(b)

Figura 3: (a) Un conjunto de puntos aleatorio. (b) La envoltura convexa de dichos puntos.

2.3. Poliedro Simple

Un poliedro es simple cuando está descrito por caras que solo se intersectan en vértices y arcos. (*Esto es lo mismo que decir que no hay una arista o cara* ***colgante***)



(a) (b)

Figura 4: (a) Un poliedro simple. (b) Un poliedro que está unido por un arco, por ende no es simple.(Imágenes generadas con Camarón)

2.4. Tetraedralización de Delaunay

Podemos expandir el concepto de Triangulación de Delaunay a 3D, en el cual los tetraedros pertenecientes a la malla tienen una circunsfera sobre la cual se posicionan todos sus vértices y los de ningún otro tetraedro, símil a la triangulación.

Una versión en 2D de un algoritmo que genera mallas de polígonos ya existe, se llama *POLYLLA* [7], el algoritmo usa un concepto similar al que se usará para esta memoria, separando triángulos en polígonos, y luego asegurándose de que estos sean simples.

2.5. Longest edge propagating path: Lepp

En dos dimensiones, *Lepp*(t) [2], el camino de propagación por el arco más largo *(por sus siglas en inglés: Longest Edge Propagating Path)* dado para un triángulo t_0 perteneciente a una triangulación $\tau = (V, E)$ con V siendo los vértices de dicha triangulación y E siendo los arcos de la misma, es una secuencia incremental de triángulos que permite encontrar un único arco más grande local en la malla (denominado terminal-edge) compartido por dos triángulos terminales (un triángulo por cada arco más grande en el terminal-edge de borde). En 3 dimensiones, *Lepp*(t) corresponde a un proceso de búsqueda multidireccional que permite encontrar un conjunto de terminal-edges.

Definición 1. E es una terminal-edge en una malla de tetraedros τ si E es el arco más largo de cada tetraedro que comparte E. Además llamamos *terminal star* (estrella terminal) $TS(E)$ al conjunto de tetraedros que comparten un terminal-edge E.

Definición 2. Para cualquier tetraedro t_0 en τ, *Lepp*(t_0) es definida recursivamente de la siguiente manera:

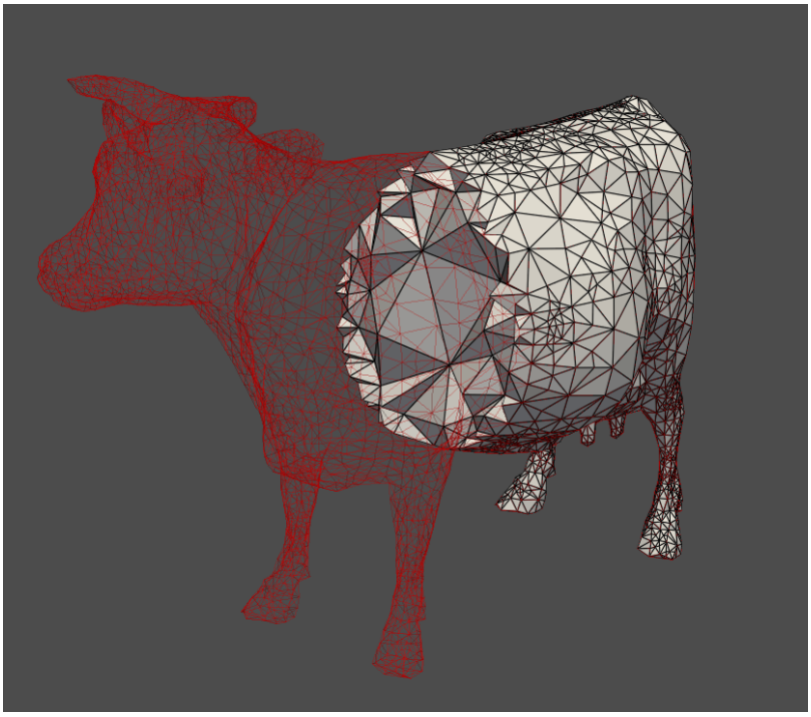


Figura 5: Tetraedralización hecha por TetGen a una malla representando una vaca, visualización con **PyVista** [9]

1. $Lepp(t_0)$ incluye a todo tetraedro t_i que comparte el arco más largo de t_0 con t, y tal que el arco más largo de t es más largo que el arco más largo de t_0.
2. Por cada tetraedro t_i en $Lepp(t_0)$, este último también contiene a todo tetraedro t que comparte el arco más largo de t_i y dónde el arco más largo de t_i es más largo que el arco más largo de t_i.

Algoritmo de bisección Lepp por un solo camino De acuerdo a la siguiente definición, se tiene un algoritmo que sigue una rama *Lepp* hasta que un terminal-edge es encontrado:

Definición 3 Para cualquier tetraedro t_0, de arco más largo L_0, se computa el algoritmo $OneBranchLepp(t_0)$ de la siguiente manera:

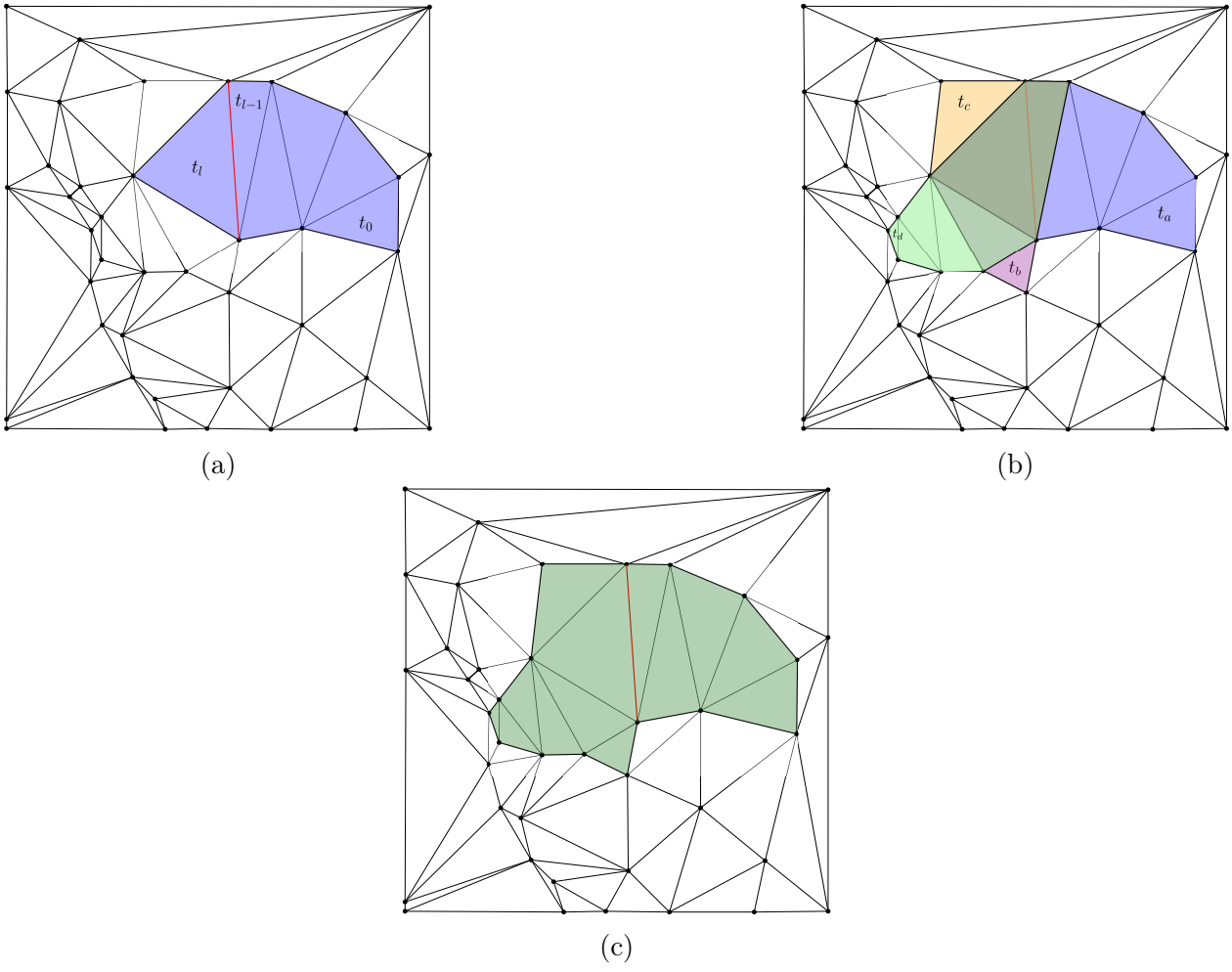
a) $OneBranchLepp(t_0)$ incluye a t_0. Luego se define el tetraedro a procesar t_{proc} igual a t_0 (de arco más largo L_{proc}).
b) Se agrega a $OneBranchLepp(t_0)$ un tetraedro t con su arco más largo L_t, seleccionado entre el set de tetraedros que comparten el arco L_{proc} y que tienen un arco $L_t > L_{proc}$
c) Repetir para t_{proc} igual a t, mientras exista un tetraedro t en el paso b.

Algorithm 1 Algoritmo de bisección Lepp por un solo camino (τ,S)

Input: τ malla de tetraedros, S set de tetraedros a ser refinados
Output: malla refinada τ_f
while $S \neq \phi$ **do**
    Por cada tetraedro $t_0 \in S$
    while t_0 permanezca en la malla **do**
        Computar OneBranchLepp(t_0), la terminal edge E y la terminal star $TS(E)$
    end while
end while

2.6. Terminal-edge regions

Una **Una terminal-edge region** R es una región formada por la unión de todos los tetraedros t_i tales que *Lepp(t_i)* comparte el mismo terminal-edge.



(a) (b) (c)

Figura 6: Terminal-edge region. **(a)** Lepp de t_0 donde el arco rojo es el terminal-edge. **(b)** Cuatro Lepps: Lepp(t_a), Lepp(t_b), Lepp(t_c) y Lepp(t_d), con el mismo terminal-edge. **(c)** Terminal-edge region generada por la unión de Lepp(t_a), Lepp(t_b), Lepp(t_c) y Lepp(t_d).

3. Estado del Arte

En la actualidad algunos de los métodos para realizar simulaciones de ecuaciones diferenciales sobre distintos objetos en 3 dimensiones utilizan mallas de tetraedros o conjuntos de puntos, entre estos están los métodos de **Elementos Finitos** y de **Diferencias Finitas**, respectivamente. No se entrará en detalle sobre las propiedades de cada uno de estos métodos, pues no son relevantes para este informe. A diferencia de los métodos ya mencionados, existe otro método, el cual es una refinación del método de **Elementos Finitos**, que se llama **Método de Elementos Virtuales**, es este método el que nos permite realizar simulaciones sobre geometrías más arbitrarias que tetraedros, como lo son las mallas de poliedros, disminuyendo el número de elementos dentro de los cuales se deben realizar los cálculos para la simulación. Debido a esto tener algoritmos para construir mallas de poliedros es de gran utilidad para aprovechar las ya mencionadas ventajas del **Método de Elementos Virtuales**.

3.1. Generadores de mallas de poliedros no tetraédricas

Algunos software conocidos para generar mallas de poliedros, no tetraédricas, se listan a continuación:

- *VoroCrust*: Un algoritmo creado en los laboratorios Sandia de Estados Unidos, necesita autorización de el mismo laboratorio para ser usado.
- *coreform Cubit THex* : Toma una malla de tetraedros y la transforma en una malla de hexaedros, técnicamente es una malla de poliedros, pero está limitada en alcance.

Para el algoritmo a desarrollar se utilizarán como input mallas que ya están compuestas de tetraedros, estas mallas estarán dadas por *TetGen*, el cual se explicará a continuación.

3.2. Generación de mallas de tetraedros usando TetGen

TetGen [8] es una aplicación que genera mallas de tetraedros a partir de conjuntos de puntos en un dominio de 3 dimensiones, esta aplicación permite la generación de mallas en dominios no convexos, las cuales se pueden utilizar para simulaciones y otras aplicaciones.

La ventaja de **TetGen**[1] es que estos puntos están hechos conforme a una tetraedralización de Delaunay, que es la ampliación a 3 dimensiones de la Triangulación de Delaunay, esto asegura que las mallas creadas no tengan poliedros altamente obtusos, además de preservar los bordes del dominio que se le da, generando mallas tetraédricas de buena calidad.

TetGen utiliza lo que se llama PLC (Piecewise Linear Complex) para generar su tetraedralización. Un PLC X es un set de celdas que satisfacen las siguientes propiedades:

1. La frontera de cada celda en X es una unión de celdas en X.
2. Si dos celdas distintas $f, g \in X$ se intersectan, su intersección es una unión de celdas en X

[1] Página web de **TetGen**

En las siguientes figuras se muestra un ejemplo de un PLC y dos ejemplos que no son PLC.

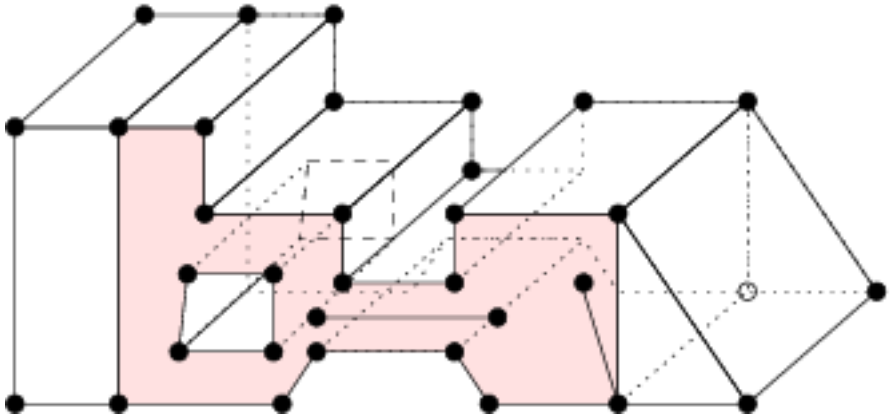
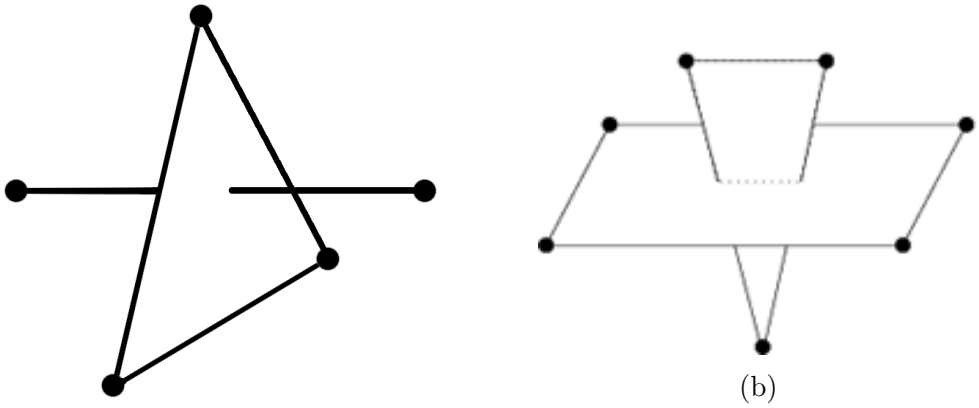


Figura 7: Un PLC, el área roja representa una cara del PLC. [8]



(a) (b)

Figura 8: En (a) la figura no es un PLC puesto que un arco atraviesa una cara, en (b) es porque una cara atraviesa a otra cara. [8]

Es necesario explicar algunos de los formatos de input y output que utiliza **TetGen** que son relevantes para esta memoria.

3.2.1. Inputs

.node*(input/output)*

Un archivo *.node* contiene una lista de puntos en 3D.

En la primera línea se tiene:
`<# de puntos> <dimensión (3)> <# número de atributos> <marcadores de frontera>`
En las líneas siguientes se tienen los puntos:
`<Punto i> <x> <y> <z> [atributos] [marcador de frontera]`

Tanto los atributos como el marcador de frontera son opcionales, por lo cual para simplicidad no se utilizaron en esta ocasión, todo esto está disponible en la siguiente página: https://wias-berlin.de/software/tetgen/1.5/doc/manual/manual006.html#ff_node

```
# Numero de nodos, 3 dim, sin atributo, sin marcador de frontera
8  3  0  0
# Indice del nodo, coordenadas del nodo x y z
1  0.0 0.0 0.0
2  1.0 0.0 0.0
3  1.0 1.0 0.0
4  0.0 1.0 0.0
5  0.0 0.0 1.0
6  1.0 0.0 1.0
7  1.0 1.0 1.0
8  0.0 1.0 1.0
```

3.2.2. Outputs

.ele ***(input/output)***

El formato *.ele* contiene una lista de tetrahedros. A continuación se deja su formato:

```
Primera Linea:
<# de tetraedros> <nodos por tetraedro (4 o 10)> <atributo de región (0 o 1)>
Las lineas que quedan listan los tetraedros:
<# tetraedro> <nodo> <nodo> ... <nodo> [atributo]
...
```

Cada tetraedro tiene 4 vértices, en el caso de que el switch -o2 sea usado, tendría 10 vértices, pero no se utilizará en este caso. Cada nodo es un índice en el archivo *.node* correspondiente.
Si el atributo de región en la primera línea es 1, cada tetraedro tiene un entero adicional en la última columna, estos atributos son principalmente usados para identificar qué tetraedro de la tetraedralización está asociado con algún sub-dominio del PLC.

Este archivo es el output por defecto de **TetGen**, puede ser omitido pero para nuestra aplicación es necesario.

.face ***(input/output)***

El formato *.face* contiene una lista de caras triangulares de la tetraedralización, de la siguiente manera:

```
Primera Línea: <# de caras> <marcador de frontera (0 o 1)>
Las líneas que quedan listan el número de caras:
<# cara> <nodo> <nodo> <nodo> ... [marcador de frontera] ...
...
```

En su forma más básica, cada cara tiene 3 vértices y posiblemente un marcador de frontera.

Los nodos al igual que en *.ele* son índices que apuntan al archivo *.node* correspondiente.

En esta memoria no se utilizarán los marcadores de frontera, pero si se utilizará el switch *-nn*. El efecto de este switch es que cada cara triangular contiene 2 índices adicionales (que vienen después del marcador de frontera) los cuales son referencias al archivo *.ele* correspondiente, estos indican los tetraedros que contienen esa cara, un -1 indica que no hay tetraedros adyacentes.

Por defecto **TetGen** solamente emite las caras que componen la envoltura convexa de la malla al archivo *.face*, pero se utilizará el switch *-f*, el cual hace que **TetGen** emita todas las caras (incluyendo caras interiores) de la tetraedrización. En este caso, cada cara interior tendrá siempre un 0 como su marcador de frontera.

.edge *(input/output)*

Un archivo *.edge* contiene una lista de arcos de la tetraedralización. El archivo está estructurado de la siguiente manera:

```
Primera línea: <# de arcos> <marcador de frontera (0 o 1)>
Las lineas restantes listan el # de arcos:
<# arco> <endpoint1> <endpoint2> ... [marcador de frontera] ...
...
```

En su forma más básica, cada arco tiene 2 endpoints y posiblemente un marcador de frontera, el cual se omitirá en esta memoria. Los endpoints son índices en el archivo *.node* correspondiente.

Además, si el switch *-nn* es usado, cada arco contiene un índice adicional, correspondiente al tetraedro que contiene el arco en el archivo *.ele* correspondiente, siendo marcados con un -1 los arcos que no pertenecen a ningún tetraedro.

3.3. Búsqueda de Vacíos

DELFIN++

El algoritmo que se implementó está basado en un programa para realizar detección de vacíos astronómicos. Los vacíos son sectores en el espacio de puntos 3D que tienen una baja densidad de galaxias. Dicho algoritmo se llama: **DELFIN**++(DELaunay edge void FINder ++), el cual usa *QHull* (Significando *Quick Hull*) [3] para generar una malla de tetraedros a partir de un conjunto de puntos, agrupando tetraedros de acuerdo a un criterio de densidad de los mismos, esto con el fin de ayudar a la detección de los vacíos mencionados anteriormente. [1] Esto lleva a la creación de poliedros que representan vacíos (Ver Figura 9)

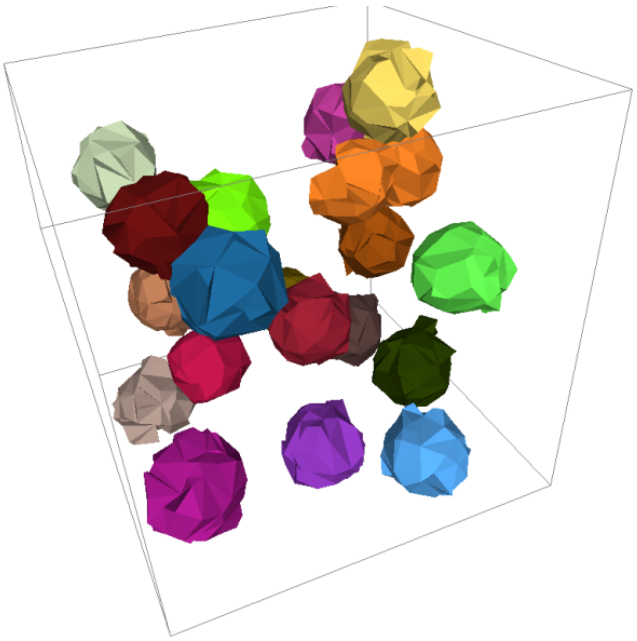


Figura 9: Ejemplo de vacíos encontrados por DELFIN++ [5]

Este subproducto de la detección de vacíos crea mallas de poliedros, que si bien en su estado actual no se pueden utilizar para la creación de mallas arbitrarias de poliedros, si sirve como punto de partida para la creación de un algoritmo que cumpla esta función, pues tiene criterios para agrupar tetraedros convirtiéndolos en mallas de poliedros. Por esto mismo es que se tomará **DELFIN**++ como base para la creación del algoritmo de generación de mallas de poliedros.

Esta limitación se debe al uso de *QHull* para la generación de mallas, ya que tiene como requerimiento que el dominio sobre el cual se va a generar la malla sea convexo. Por este motivo se cambiará a *TetGen*, pues este último permite generar una malla de tetraedros en el interior de cualquier superficie cerrada no necesariamente convexa. Esto quiere decir que dentro de la superficie cerrada pueden existir agujeros o polígonos que separen la superficie en regiones de distintos materiales. Usar *TetGen* permitirá generar mallas para dominios con geometrías complejas.

A continuación se deja el algoritmo **DELFIN**++ como pseudocódigo:

Algorithm 2 Algoritmo DELFIN++ de búsqueda de vacíos por densidad de puntos

Leer la teselación de Delaunay;
Leer **densidad_umbral**;
Leer **volumen_umbral**;
Calcular densidad por punto;
Ordenar puntos por densidad ascendente;
lista_vacios = $\varnothing$;
for each punto p en la lista de puntos ordenada **do**
    if la densidad de p es mayor a **densidad_umbral then** break;
    end if
    nuevo_vacio = **True**;
    Obtener los tetraedros $t_1, ..., t_n$ a los que pertenece p;
    tetraedros_p = $[t_1, ..., t_n]$
    for each tetraedro t en **tetraedros_p do**
        if t pertenece a un vacío v de **lista_vacios then**
            for each tetraedro r en **tetraedros_p do**
                if r no es parte de un vacío **then**
                    Agregar r al vacío v;
                else if r pertenece al vacío w, $w \cap v = \varnothing$ **then**
                    $v = v \cup w$;
                    lista_vacios = **lista_vacios**$\backslash w$;
                end if
                nuevo_vacio = **False**;
                break;
            end for
        end if
    end for
    if nuevo_vacio then
        lista_vacios $\cup$ poliedro**tetraedros_p**;
    end if
end for
Eliminar de **lista_vacios** a aquellos vacíos con volumen menor a **volumen_umbral**, con un arco en el borde de la cobertura convexa o con 90 % de tetraedros en su superficie de mala calidad;

La motivación de este trabajo es automatizar el proceso de generación de mallas poliédricas y evaluar dichas mallas en simulaciones, viendo si estas se pueden realizar de manera más rápida que utilizando otro tipo de mallas.

3.4. Generación de mallas de polígonos

POLYLLA

El algoritmo *POLYLLA* [7] (Polygonal meshing algorithm based on terminal-edge regions) sirve para construir mallas poligonales arbitrarias, con condiciones de Delaunay. Este algoritmo corre en 3 fases:

1. Fase de etiquetado: Cada arco (arista) es etiquetado como más largo, terminal o de frontera para construir terminal-edge regions. El algoritmo también etiqueta un triángulo en cada terminal-edge region como triángulo semilla utilizado en la siguiente fase para construir cada polígono.
2. Fase de recorrido: Consiste en la generación de polígonos desde triángulos semilla. En esta fase los arcos-frontera de un terminal-edge region se guardan en sentido contrario a las agujas del reloj (counter clock-wise o ccw) para construir un polígono. Los polígonos generados que no son simples pasan a la fase de reparación.
3. Fase de reparación: Los polígonos con arcos-barrera se particionan para llegar a polígonos simples.

Las figuras 10, 11, 12 muestran la fase de etiquetado, de recorrido y de reparación, respectivamente.

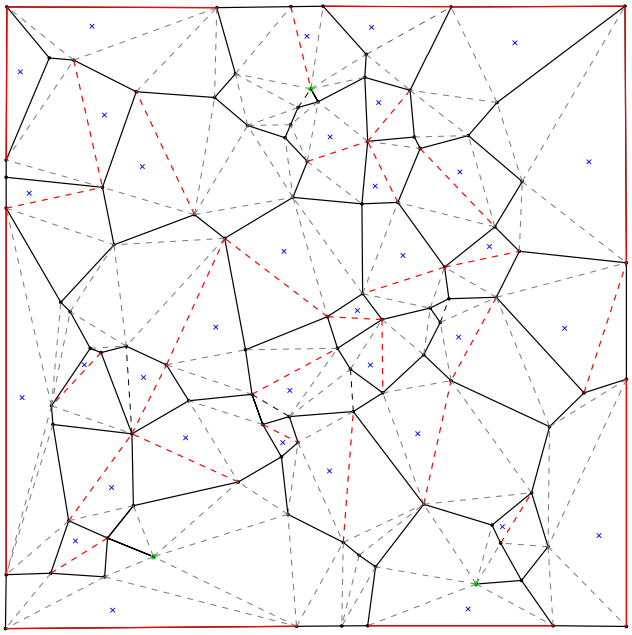


Figura 10: Triangulación etiquetada generada durante la fase de etiquetado. Las líneas sólidas son arcos frontera, las líneas punteadas son arcos internos y las líneas rojas punteadas son terminal-edges y terminal-edges de borde. Triángulos con una cruz azul son triángulos semilla. [7]

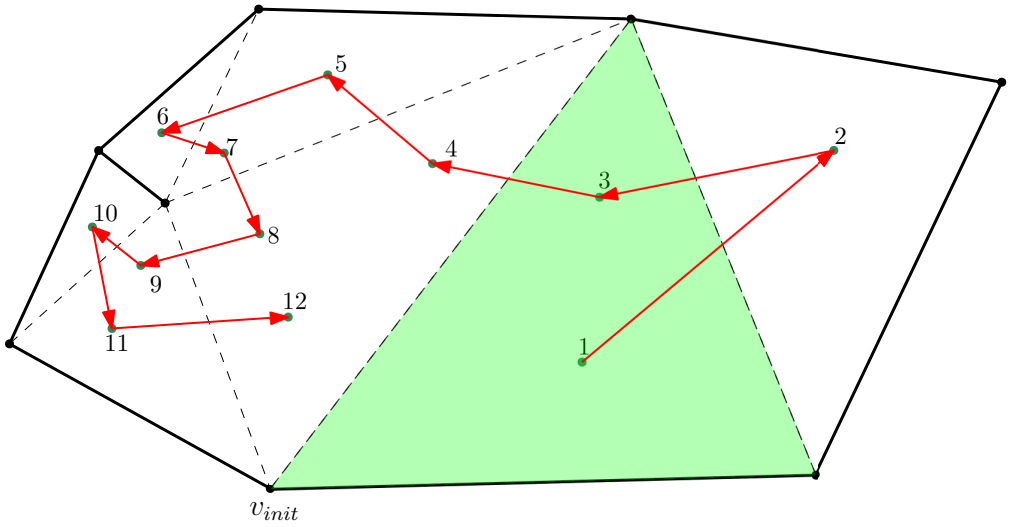


Figura 11: Fase de recorrido. El triángulo verde es el triángulo semilla de esta *terminal-edge region*. Los números indican en qué orden fue visitado cada triángulo. [7]

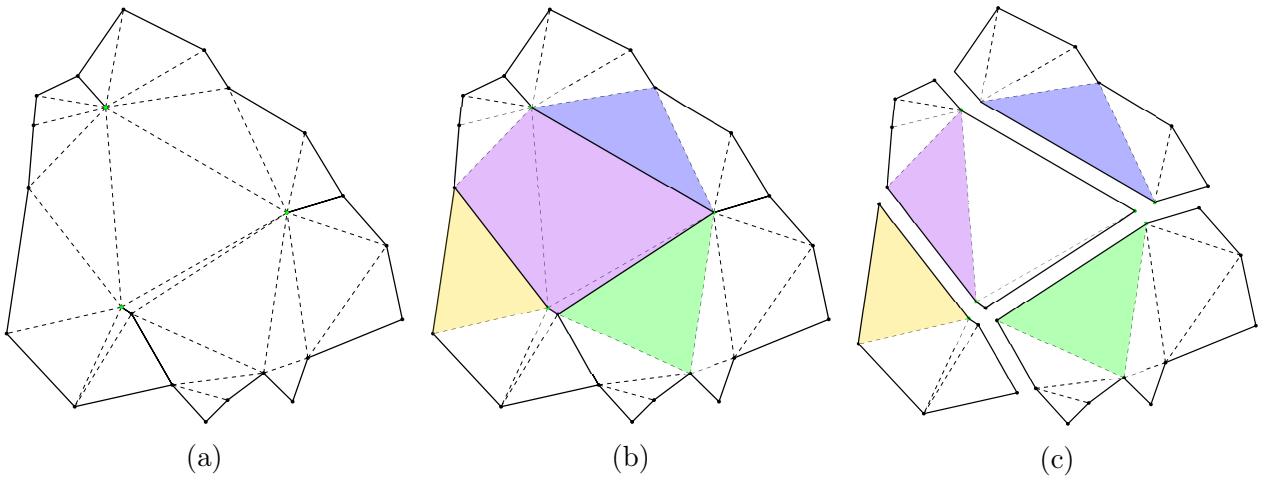


Figura 12: Ejemplo de una división de un polígono no simple usando puntas de arcos barrera. (a) Polígono no simple. (b) Arcos internos del medio que tocan puntas de arcos barrera marcados como arcos frontera (línea sólida) y triángulos semillas (en colores) guardados en la lista semilla L_p y marcados. (c) Los cuatro nuevos polígonos sin puntas de arcos barrera. (vértices colgantes) [7]

3.5. Formatos

ViSF

El formato de salida del algoritmo creado es **ViSF**, es un formato que puede ser tanto binario como ASCII, en el cual se pueden expresar un conjunto de puntos, las caras que se tienen en la malla, y los poliedros que la conforman, también es posible poner las relaciones de vecindad entre los mismos, que en el caso del trabajo desarrollado, se omitieron.

La primera línea de un archivo **ViSF** especifica el endianness del archivo, siendo 0 big endian y 1 little endian, con 2 se puede especificar que el archivo está en formato ASCII, y es lo que se usó en este caso. Además especifica el tipo de malla que se guardó en el archivo con un entero, siendo 0 una nube de vértices, 1 malla de polígonos y 2 malla de poliedros.

A continuación se deja el layout de un archivo **ViSF**:

```
<Endianness (char ASCII)> <Tipo de malla (int)>
<# de vértices>
<x> <y> <z>
...
<# de polígonos (caras)>
<# de vértices del polígono 0> <id vértice 0> <id vértice 1> ...
...
<Presencia de relaciones de vecindad (char)>
<# de vecinos> <id vecino 0> <id vecino 1>
...
<# de poliedros>
<# de caras> <id cara 1> <id cara 2> ...
...
```

Este formato se utilizará como output del programa, ya que permite definir a base de triángulos varios poliedros para ser renderizados después con otro programa llamado **Camarón** [4], programa para el cual fue desarrollado el formato.

4. Análisis y diseño

Métodos nuevos tales como el Método de elementos virtuales (*VEM* por sus siglas en inglés) pueden utilizar cualquier tipo de poliedro como su elemento básico, mejorando la velocidad de las simulaciones realizadas con el método con respecto a otros debido a la menor cantidad de celdas y puntos de malla utilizados para resolver el mismo problema, en comparación a otros métodos tales como el método de diferencias finitas y el método de elementos finitos.

Debido a que *VEM* es relativamente nuevo, no existe un algoritmo automático para generar mallas usando poliedros arbitrarios. Actualmente *VEM* es usado con mallas poligonales formadas por celdas de Voronoi, pero este tipo de mallas no nos permiten explorar todo el potencial del método, así que existe espacio para investigar sobre nuevos algoritmos que generen mallas de poliedros.

Muchos ejemplos de uso utilizan dominios hexagonales o cuasi-hexagonales por el hecho de que existen herramientas para generar este tipo de mallas, en la figura 13 se puede apreciar un ejemplo de dicho uso.

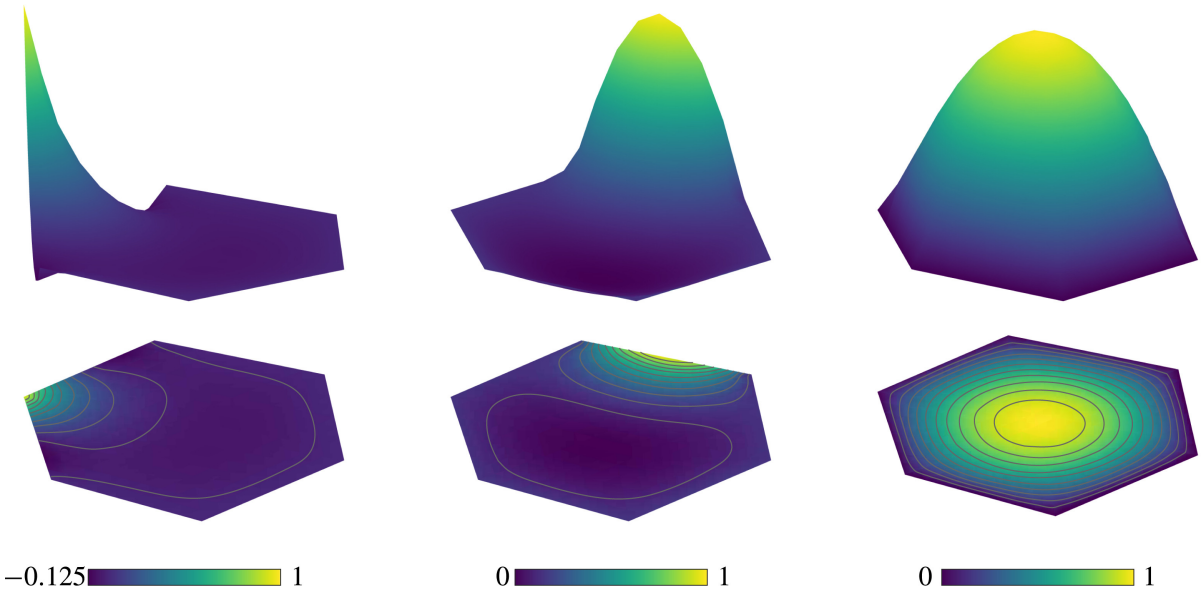


Figura 13: Ejemplo de uso del método de elementos virtuales para resolver un problema de condición de frontera en un dominio hexagonal [6]

Basado en esta necesidad de un generador de mallas de poliedros, adaptable a cualquier dominio geométrico, se pretende desarrollar un nuevo generador de dichas mallas que utiliza el concepto de **Terminal-edge regions** [7], relacionado con *Lepp* (explicado anteriormente).

Se pretende solucionar este problema mediante la implementación de un algoritmo que tome mallas generadas por **TetGen** y a partir de esas mallas tetraédricas genere mallas poliédricas, en esta ocasión se buscan mallas poliédricas con caras triangulares. Esta solución inicialmente estuvo inspirada en el algoritmo *DELFIN++*, por razones que se explicarán más adelante, se pasó al algoritmo DELFIN y luego se construyó con elementos de *POLYLLA*.

4.1. Requerimientos

Para generar una solución satisfactoria al problema plateado se requiere que el algoritmo y su implementación tengan las características descritas a continuación:

1. La solución debe ser capaz de tomar mallas tetraédricas generadas desde *TetGen*.
2. Se necesita tener un archivo de salida que permita interpretar correctamente los poliedros como tales.
3. El algoritmo no debe introducir nuevos puntos a la malla original.
4. La solución debe ser capaz de procesar mallas con una gran cantidad de puntos (mayor a 10^5).

4.2. Funcionalidad

Para cumplir los requerimientos se implementarán las siguientes funcionalidades:

- Lectura de archivos de Input de TetGen.
- Guardado en memoria de dichos archivos.
- Procesamiento de mallas conservando los puntos en estructuras dentro de memoria.
- Almacenamiento de la malla en formato *ViSF*.
- Una sola referencia por cada punto procesado (evitar duplicación).

5. Implementación

5.1. Primeros acercamientos

En una primera instancia se intentó adaptar *DELFIN++* para utilizar la fase de creación de subvacíos como poliedros de una malla, pues estos son básicamente poliedros que no contienen puntos ni caras interiores. Esta primera aproximación falló debido a que el criterio de densidad que ocupa *DELFIN++* si bien es muy bueno para la búsqueda de vacíos, no se presta de buena manera para la creación de poliedros, ya que el algoritmo puede dejar puntos dentro de los vacíos si es que su densidad es lo suficientemente baja. Además se requiere eliminar los volúmenes y densidades umbrales para obtener una poliedrización, lo que en realidad no funciona, puesto que termina siendo una tetraedrización con vacíos en forma de tetraedro. Por esto se hubiese necesitado una refactorización de prácticamente toda la base de código para acomodarla al problema.

En cambio *DELFIN* ocupa el criterio de unión mediante el arco más largo que compartan los subvacios, lo que se presta bien para una conversión a creación de poliedros desde subvacíos, pero como se discutió anteriormente, la implementación del algoritmo en C++ contiene código con una complejidad ciclomática alta (entre 10-15) como se explica en la memoria de Valeria Guidotti [5], el cual es muy difícil de adaptar.

A continuación se deja el algoritmo **DELFIN**:

Algorithm 3 Algoritmo DELFIN de búsqueda de vacíos por arco más largo

```
1: Leer la teselación de Delaunay;
2: Leer volumen_umbral;
3: lista_vacios = ∅;
4: Ordenar tetraedros por arco más largo;
5: repeat
6:     if el arco más largo del siguiente tetraedro sin visitar es más corto que largo_umbral
   then
7:         break;
8:     end if
9:     if el arco más largo del siguiente tetraedro sin visitar es el más largo de los tetraedros
   que lo comparten then
10:        obtener todos los tetraedros t_1, ..., t_n que comparten ese arco;
11:        set_tetraedros = [t_1, ..., t_n];
12:        marcar todos esos tetraedros como usados;
13:        for each tetraedro t vecino de set_tetraedros do
14:            if t comparte su arco más largo con un tetraedro de set_tetraedros then
15:                set_tetraedros = set_tetraedros ∪ t;
16:                marcar t como usado;
17:            end if
18:        end for
19:        if el volumen de los tetraedros en set_tetraedros ≥ volumen_umbral then
20:            lista_vacios = lista_vacios ∪ poligono(set_tetraedros);
21:        end if
22:    end if
23: until no hay tetraedro que procesar
24: for each arco e en set_arcos do
25:    if el largo de e es mayor a largo_umbral then
26:        Agregar el subconjunto de prevacios de lista_vacios que comparten a e;
27:    end if
28: end for
29: Eliminar de lista_vacios a aquellos vacíos con un arco en la frontera;
```

Como se puede apreciar, en la línea 4 se ordenan los tetraedros que le pasan al programa por orden de arco(arista) más largo, esto es muy importante porque permite utilizar *Lepp* para recorrer los tetraedros.

En cualquiera de los casos anteriores, se buscó la manera de extraer la fase de generación de subvacíos y dejar de lado la unión de subvacíos, pues estos mismos son los poliedros que nos interesa generar, y el resultado final es la malla completa, independientemente de la forma que tenga la misma, el paso de un algoritmo con criterios geométricos para un uso astronómico a un algoritmo para generación de mallas se estimaba que fuese el trabajo de esta memoria.

Esto no fue así, por las razones anteriormente mencionadas se descartó tanto *DELFIN++* como *DELFIN*. Este último también hubiese requerido una reingeniería substancial de la base de código, se intentó adaptar solamente las partes que correspondían a la construcción de subvacíos y la unión de los mismos, en particular `VoidBuilder, VoidJoiner y EdgeJoiner`, los cuales tienen una complejidad de 6, 10 y 15, respectivamente como se puede apreciar en la siguiente tabla. [5]

Name	**Avg Stmts/Method**	**Max Complexity**	**Max Depth**	**Avg Depth**	**Avg Complexity**
Dictionary	2.8	3	2	0.96	1.5
PointDictionary	5.8	5	5	1.48	1.83
Edge	1.7	2	2	0.74	1.07
EdgeDictionary	4.7	3	2	1.15	1.29
Facet	3.1	4	4	1.16	1.42
FacetDictionary	10.3	15	7	1.93	3.17
VoidBuilder	17.7	6	4	2.24	3.67
DelfinBuilder	11.0	6	4	2.24	3.67
VoidJoiner	21.7	10	5	2.62	5
EdgeJoiner	10.8	15	5	1.67	5.20
VoidSpace	1.6	3	3	0.80	1.36
VoidContainer	2.1	2	2	0.67	1.14

DELFIN a pesar de representar un algoritmo muy similar al que se esperaba obtener, cae corto en temas de implementación, pues hay mucho código que no se utilizará, ya sea porque corresponde a la fase de agrupación de vacíos u otras consideraciones para el uso astronómico del software, o porque simplemente existían partes del código que tenían una arquitectura incompleta (motivo por el cual se comenzó por intentar adaptar *DELFIN++*).

3D POLYLLA Face

En consideración del tiempo de implementación, y la necesidad de tener un prototipo utilizable para realizar la validación de la idea inicial del algoritmo, se pasó a utilizar como base un programa hecho en el lenguaje de programación **Python** por Sergio Cuyi. Este representa una buena base puesto que es de implementación concisa, lo que permite utilizar casi todas las clases ya disponibles, y simplemente agregarle funcionalidad, este algoritmo corresponde a *3D POLYLLA Face*.

Como su nombre lo indica, *3D POLYLLA Face* es una extensión del algoritmo en 2D, por lo cual para entenderlo basta entender cada uno de sus pasos como símiles en 3 dimensiones, en vez de triángulos se ven tetraedros, y en vez de polígonos se construyen poliedros, además de unir los tetraedros por caras en vez de aristas. Los pasos entonces quedarían de la siguiente manera:

1. Fase de etiquetado: Cada arco (arista) es etiquetado como más largo, terminal o de frontera para construir terminal-edge regions. El algoritmo también etiqueta un tetraedro en cada terminal-edge region como tetraedro semilla utilizado en la siguiente fase para construir cada poliedro.
2. Fase de recorrido: Consiste en la generación de poliedros desde tetraedros semilla. En esta fase los arcos-frontera de un terminal-edge region se guardan para construir un poliedro. Los poliedros generados que no son simples pasan a la fase de reparación.
3. Fase de reparación: Los poliedros con arcos-barrera se particionan para llegar a poliedros simples.

Cabe destacar que en este caso (en 3 dimensiones) una **terminal-edge region** corresponde a una región de tetraedros que comparten un mismo arco.

5.2. Reingeniería a *3D POLYLLA Face*

La implementación del algoritmo no estaba completa, además de ser distinta al algoritmo deseado, pues se unen tetraedros a partir de caras y no a partir de arcos. Estaba la fundación que permitía leer archivos generados desde *TetGen* y pasarlos a estructuras de datos en Python, esto fue un buen primer acercamiento ya que permitió generar modificaciones a dichas estructuras y agregar funcionalidad con la finalidad de tener un algoritmo que cumpla con los objetivos de la memoria.

Primeramente se tenían estructuras de datos existentes, a las cuales solamente se les hicieron modificaciones, se listan a continuación:

- **Vertex**: Representa a un punto en el espacio 3D, posee sus coordenadas (x, y, z) y un índice i.

- **Face**: Representa una cara triangular, posee sus vértices, sus arcos, y referencias a sus caras vecinas, además contiene un booleano para saber si pertenece a la frontera de la malla o no.

Luego están las nuevas estructuras de datos que fueron implementadas, las cuales se ex-

plican a continuación:

- **Edge**: Representa un arco, al igual que *Vertex*, tiene un índice, además de 2 referencias a los puntos que demarcan al arco, además de una lista de los tetraedros que comparten el arco, esta lista es muy importante para la implementación del algoritmo pues nos permite evitar reutilizar tetraedros.

- **Tetrahedron**: Representa un tetrahedro, almacenando los cuatro vértices que posee como índices (encontrados en la estructura *Vertex*. Además posee una lista de las caras que lo conforman, un *set* de los arcos que posee y dos booleanos, uno para saber si es un tetrahedro de borde y otro para saber si el tetrahedro fue ya visitado por el algoritmo o no.

- **Polyhedron**: Representa a un poliedro, contiene una lista de las caras que lo conforman.

- **TerminalEdge**: Representa un terminal-edge, contiene un diccionario con los tetrahedros que lo comparten, la llave es el índice del tetrahedro.

5.3. Algoritmo

El algoritmo implementado consta de varias fases, las cuales se detallan a continuación.

Primero se leen los archivos generados por *TetGen*, en particular los archivos `X.node`, `X.ele`, `X.face`, `X.edge`, donde X es el nombre del archivo que se le pasó a *TetGen*, estos archivos se guardan en las respectivas estructuras de datos.

En segundo lugar se guardan las referencias entre estructuras para rellenar los campos que el paso de lectura no pudo rellenar.

En tercer lugar se calculan los largos de cada arco, con eso se obtienen los arcos más largos de cada tetraedro y se guardan los largos de los arcos en la estructura **Edge**.

En cuarto lugar se realiza una búsqueda *Depth-First Search* sobre todos los tetraedros de la malla, en la cual se empieza por los tetraedros que contienen al arco más largo, se recorren esos tetraedros marcándolos como visitados y no considerándolos si ya han sido visitados, estos tetraedros conforman un terminal-edge, y por cada tetraedro se toma su arco más largo y se continúa el DFS, esto produce una terminal-edge guardada en la estructura `TerminalEdge`.
Este paso se repite por cada uno de los arcos, partiendo desde el arco más largo de la malla y siguiendo en orden descendente.

Por último el conjunto de terminal-edges generado en el paso anterior se convierte a poliedros de la estructura `Polyhedra`, lo que permite su exportación en formato *ViSF*. Además durante la conversión se elimina cualquier poliedro que no contenga tetraedros, esto es equivalente a que el poliedro no contenga caras.

Podemos separar esto en 3 pseudoalgoritmos, el primero combina las primeras 3 fases, el

segundo es el DFS que se realiza sobre los tetraedros de la malla, y el 3ro es la generación de poliedros a partir del resultado de este DFS.

Algorithm 4 Lectura e inicializacion de estructuras

```
Leer los archivos .node .ele .face y .edge
Rellenar referencias a estructuras Edge, Face, Tetrahedron
for each Edge e do
    Calcular y guardar largo de e
end for
lista_edges = lista ordenada de arcos de mayor a menor largo.
return lista_edges
```

Algorithm 5 DFS sobre malla de tetraedros

```
Inicializar lista_terminal_edge
for each Edge e ∈ lista_edges do
    Inicializar terminal_edge
    for each Tetrahedron t ∈ e do
        if t no ha sido visitado then
            Añadir t a terminal_edge
            Marcar t como visitado
            Llamar a DFS sobre malla de tetraedros con el siguiente arco de mayor largo de t que no haya sido utilizado
        end if
    end for
    Añadir terminal_edge a lista_terminal_edge
end for
return lista_terminal_edge
```

Algorithm 6 Generador de malla de poliedros

```
Inicializar lista_poliedros
for each TerminalEdge te ∈ lista_terminal_edge do
    Inicializar poliedro
    for each Tetrahedron t ∈ te do
        for each cara f de t do
            if La cara de t no ha sido usada then
                Añadir la cara de t a poliedro
                Marcar la cara como usada
            end if
        end for
    end for
    Añadir poliedro a lista_poliedros
end for
Eliminar todos los poliedros vacíos de lista_poliedros
return lista_poliedros
```

5.4. Detalles de implementación

El programa fue desarrollado en Python usando como base el código de *3D POLYLLA Face* explicado anteriormente, este código utiliza solamente librerías estándar de Python, por lo cual puede ser ejecutado en cualquier sistema operativo que soporte Python 3.8 o superior (es recomendado Python 3.10). Se aplicaron extensas modificaciones, utilizando solamente un par de estructuras ya existentes en el algoritmo inicialmente tomado para leer archivos desde TetGen. El algoritmo generado fue hecho a partir de 0.

Además el programa utiliza de manera extensa las clases de python para sus estructuras internas, lo cual se podría mejorar con el uso de *dataclasses* de python para disminuir el consumo de memoria, pero ese cambio no está presente en la versión presentada acá. El programa puede ser llamado a través de la línea de comandos con la instrucción `python3 polymesh.py <ARCHIVO>`, siendo <ARCHIVO>el nombre del conjunto de archivos que generó tetgen antes del sufijo de archivo, por ejemplo:

Si se tiene: `malla.node malla.ele malla.face malla.edge`, el comando para llamar al programa sería: `python3 polymesh.py malla`, ya que el programa busca los archivos por su cuenta.

5.5. Funcionalidad

A parte del algoritmo, el programa posee un par de utilidades que sirven para diversas cosas, se listan a continuación:

- **Exportador a `ViSF`**: El programa posee una función para exportar la malla de poliedros a formato `ViSF`, esta función ocupa la codificación `ASCII` de dicho formato, guardando el resultado en un archivo del nombre que se le especifique, dándole automáticamente el formato de malla de poliedros con caras triangulares.

- **Exportador a `OFF`**: El programa también posee una función para exportar a OFF, en la cual se pierden los poliedros y simplemente imprime las caras como una malla de tetraedros, esto sirve con fines de debugging.

- **Calculador de característica de Euler**: Se incluye una función que permite calcular la característica de euler, esto es, la ecuación $V - E + F = 2$ para los poliedros, esto permite saber si son poliedros no simples (aunque no captura todos los escenarios) pero si la función da falso, entonces es seguro que el poliedro no es convexo, no así si la función es verdadera.

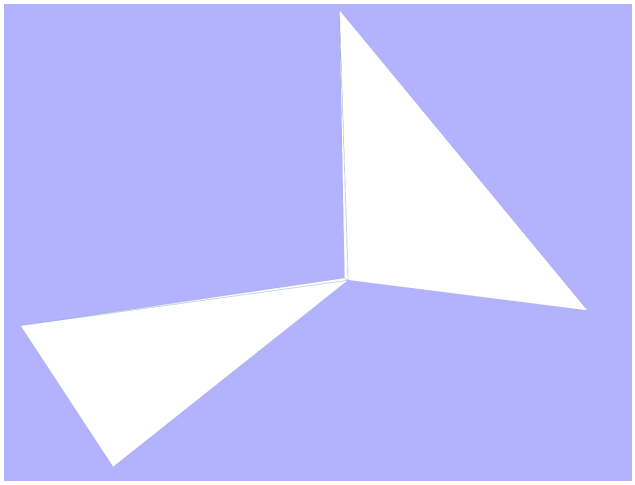
5.6. Limitaciones

La implementación presenta dos limitaciones mayores en su funcionamiento.
La primera limitación es que el algoritmo necesita que *TetGen* sea llamado con las opciones `-fznne` para que los archivos resultantes funcionen con el programa, esto pues el archivo *.edge* que se genera al utilizar estas opciones posee uno de los tetraedros que lo contiene, lo que podemos utilizar de base para hacer DFS. Además hace que el archivo *.face* contenga las caras vecinas de cada cara, con lo cual podemos ver de manera más fácil los tetraedros colindantes.

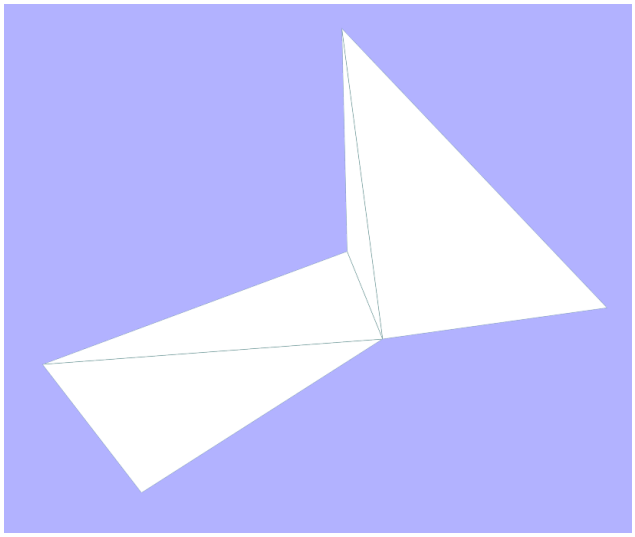
Sin estas opciones, la implementación hubiese sido considerablemente más difícil, pero como actualmente no está integrado *TetGen* en el programa en sí, se considera una limitación, ya que el usuario tiene que hacer un paso más con la malla para poder utilizar el programa.

La segunda limitación es que actualmente el algoritmo no posee fase de división de poliedros no simples (reparación de la malla), por lo cual algunas mallas quedan con imperfecciones que las hacen incompatibles con métodos numéricos, esto se puede solucionar implementando una fase de reparación, lo que no se pudo debido al tiempo dedicado al resto de la memoria. Estos poliedros no simples generalmente quedan en forma de poliedros conectados por un arco, o poliedros con caras colgantes.

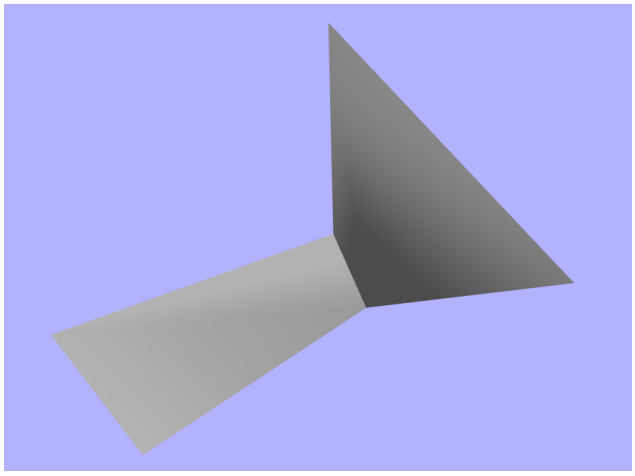
En la siguiente figura se muestran ejemplos de poliedros no simples que fueron generados por el programa.



(a)



(b)



(c)

Figura 14: Ejemplo de poliedro no simple, tanto (a) como (b) muestran que el poliedro está unido por un arco al medio, en (c) se muestra el poliedro desde el mísmo ángulo que (b) con shading para mejor apreciación.

6. Validación

El sistema que se utilizó para hacer las pruebas que se mostrarán a continuación tiene las siguientes características:

- CPU: AMD Ryzen 5 5600X (6 núcleos, 12 Hilos)
- RAM: 32GB DDR4 3200Mhz
- GPU: AMD Radeon 6900XT (16GB VRAM)
- Sistema Operativo: Ubuntu 20.04 LTS

Cabe destacar que todas las renderizaciones de poliedros se hicieron con Camarón, utilizando el formato ViSF.

A continuación se muestra una malla de poliedros, cabe destacar que es igual en forma a la malla de tetraedros correspondiente, puesto que recupera de manera íntegra sus caras, simplemente formando poliedros en vez de tetraedros, se han marcado algunos tetraedros en la figura usando Camarón.

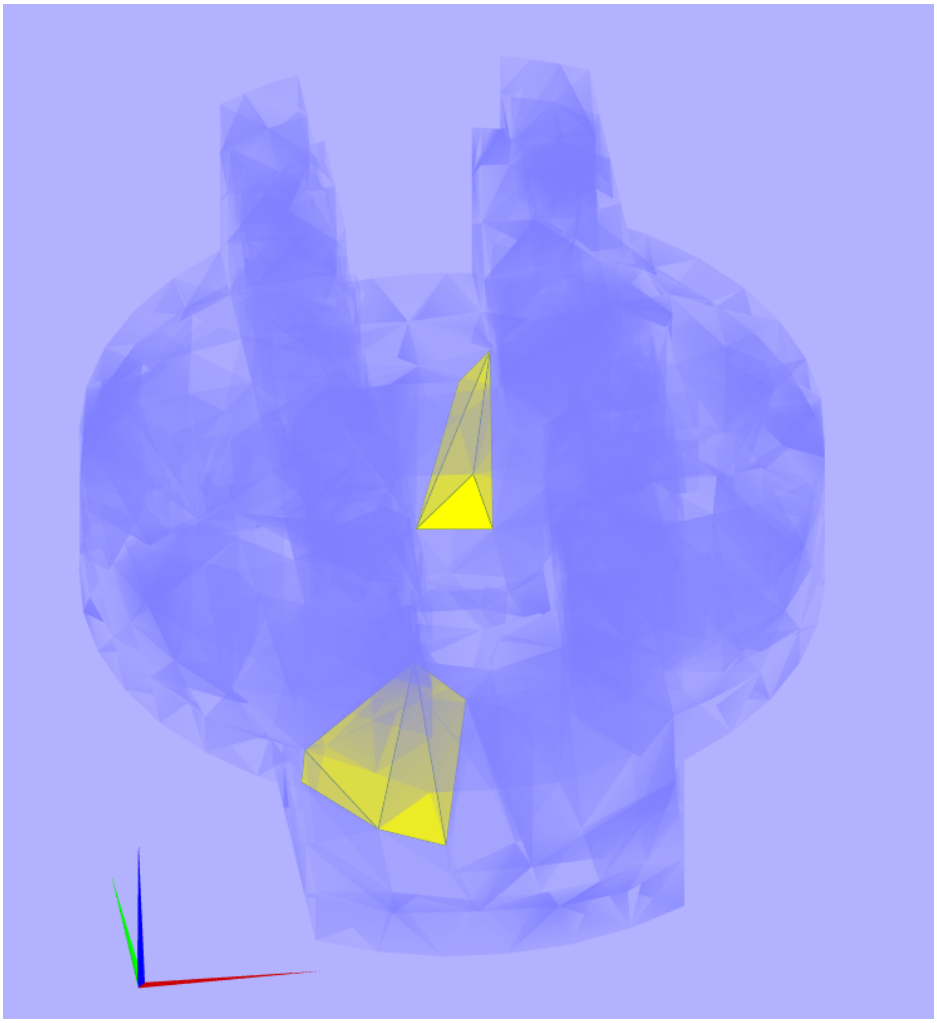


Figura 15: Malla de poliedros representando a un tornillo, se muestran en amarillo los poliedros 0 y 1, en los cuales se aprecia que tienen más caras de las que tendría un tetraedro.

El siguiente gráfico representa los tiempos de ejecución según el número de puntos, al ser este un algoritmo de un solo hilo de ejecución, este se beneficia del rendimiento *single core* de la CPU más que del rendimiento *multi-core*

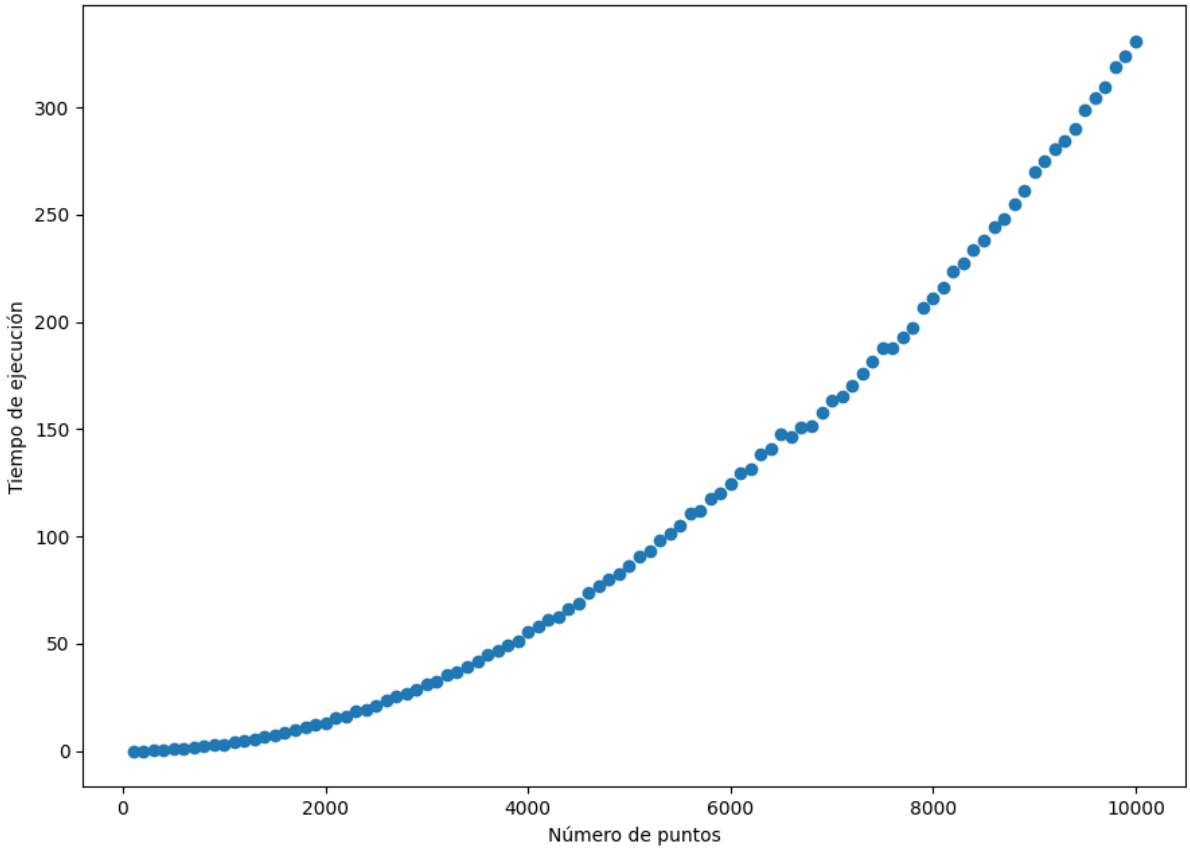


Figura 16: Gráfico del número de puntos en el input vs el tiempo de ejecución, los tiempos están en segundos.

Como se puede apreciar, el tiempo de ejecución sube de manera cuadrática con el número de puntos que posee la malla, nominalmente el dfs tiene un coste lineal, pero al crear una lista de los tetraedros visitados, estamos introduciendo dos visitas por tetraedro, una para marcar y otra cuando ya ha sido marcado, por esto mismo se tiene que el algoritmo en su completitud es más parecido a $O(n^2)$ tomando los datos empíricos

El siguiente gráfico muestra el número de poliedros no simples con respecto al número de puntos totales, cabe destacar que es una métrica que toma la cota inferior de poliedros no simples, puesto que las métricas implementadas no permiten descubrir todos los poliedros no simples, como se dijo anteriormente.

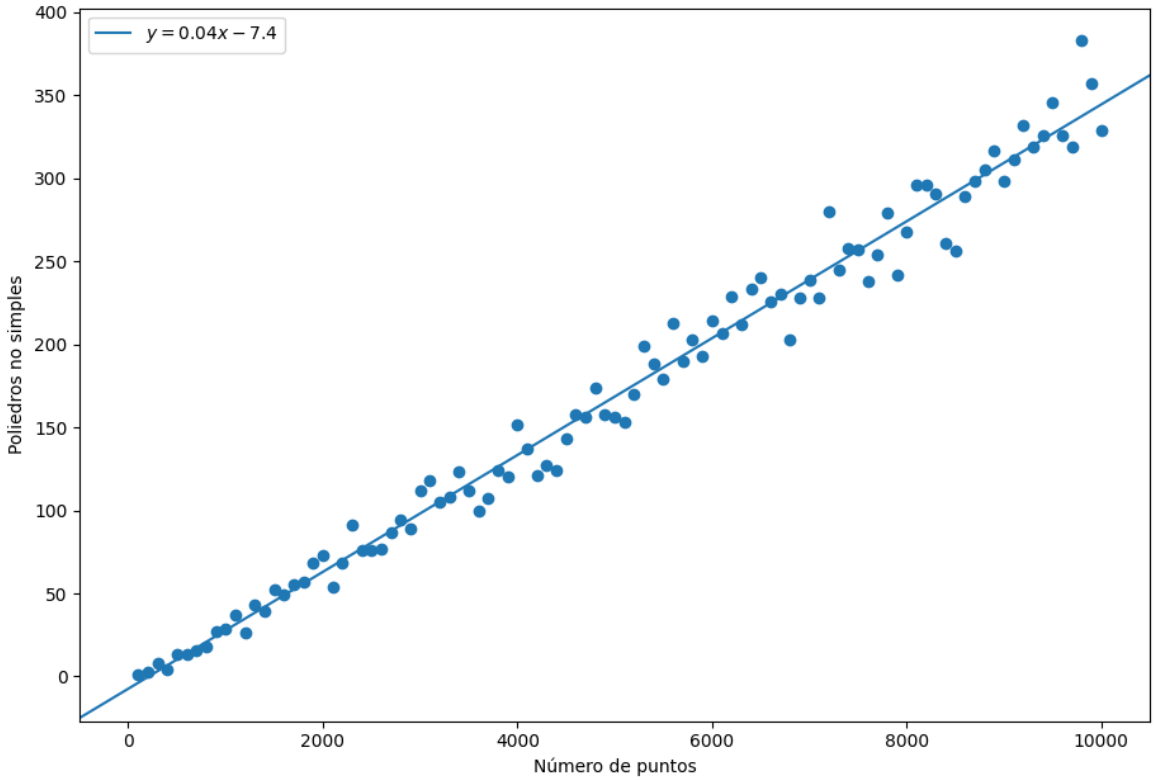


Figura 17: Gráfico del número de puntos en el input vs la cantidad de poliedros no simples que están en la malla generada

Se ve que para un conjunto de puntos aleatorios, la cantidad de poliedros no simples escala con el número de puntos, pero no de manera tan drástica como lo haría sugerir el gráfico, ya que para el conjunto de 10000 puntos se generaron 38311 poliedros, de los cuales (al menos) 347 son no simples, esto representa una proporción del 1,1 % de poliedros simples a poliedros no simples, lo que nos lleva a decir que el rendimiento del algoritmo en cuanto a calidad de mallas es bueno.

# de puntos	# de tetraedros	# total de poliedros	# poliedros no simples	# de tetraedros promedio por poliedro	promedio de caras por poliedro
100	506	313	1	1.56	5.11
1000	6312	3740	29	1.63	5.26
2000	7549	12967	73	1.66	5.32
3000	19624	11398	112	1.72	5.32
4000	26333	15277	152	1.72	5.32
5000	33057	19165	156	1.72	5.32
6000	39640	22979	214	1.73	5.32
7000	46326	26735	239	1.73	5.32
8000	52968	30549	268	1.73	5.32
9000	59704	34433	298	1.73	5.32
10000	66380	38361	329	1.73	5.32

Tabla 1: tetraedros, poliedros, poliedros no simples y tamaño promedio de los poliedros generados (en número de tetraedros) por cada set de puntos

La tabla anterior posee el número de puntos, el número de tetraedros totales para la malla original, el número de poliedros de la malla resultante, el número de poliedros no simples que tiene dicha malla y el número de tetraedros que tiene el poliedro promedio, esto no es un número entero puesto que se pierden las caras internas que están en la unión de los tetraedros para formar el poliedro deseado.
Cabe destacar que el promedio de caras por poliedro es una mejor métrica debido a estas pérdidas de caras, como podemos ver se tienen en promedio alrededor de 5 caras por poliedro generado, lo que se traduce en una reducción de elementos entre 1.5 y 1.7 veces el número original de elementos (tetraedros versus poliedros).

7. Conclusiones

La generación de mallas de poliedros es un tema bastante complejo, por lo mismo no se ha abordado mucho y tampoco existen muchos formatos que permitan la interpretación de estas mallas.

Este trabajo fue realizado bajo un contexto en el cual existía un ecosistema casi exclusivamente realizado por miembros de la Universidad de Chile para trabajar con estas mallas, ya que tanto el visualizador Camarón, como el formato ViSF, y los primeros acercamientos a un generador de mallas de poliedros fueron gracias al trabajo anterior de estudiantes y profesores. Por lo cual se tenían las herramientas necesarias para crear el algoritmo.

Si bien el trabajo no cumplió cien por ciento con lo que se quería desde un principio, hubo mucho aprendizaje, tanto mío como de las personas que me acompañaron en este proceso. El algoritmo cumple el objetivo de ser un mallador de poliedros, además de utilizar herramientas ya existentes como punto de partida, como es el caso de TetGen. Esto es un logro importante, aunque la fase de reparación de mallas no se haya podido implementar, lo que conllevó a que no se pudiese testear con VEM.

A modo de conclusión, el algoritmo cumplió los objetivos principales, sirve como demostración y como punto de partida para quien quiera refinar y mejorar la usabilidad del mismo, se conocen sus alcances, el trabajo con poliedros se complica al liberar la cantidad de caras que puede poseer un poliedro, aún así se puede explorar la posibilidad de caras poligonales además de tener una cantidad arbitraria de caras, el área todavía tiene mucho para explorar.

7.1. Trabajo Futuro

Queda como trabajo futuro la siguiente lista:

- Integración con **TetGen**: Esto implica la integración del módulo de TetGen para Python dentro del mismo programa, con lo cual se eliminaría la necesidad de preprocesamiento de la malla de parte del usuario.
- Multi procesamiento: Ya sea con la librería *Threading* o con *async*, este problema podría beneficiarse del multiprocesamiento, esto solamente para la creación de estructuras internas, ya que el problema en sí no es paralelizable.
- Fase de reparación completa: Este es probablemente el punto que más ayudaría en la utilidad del programa, crear una fase de reparación que utilice varios criterios para encontrar los poliedros no simples y/o bien separarlos en el caso de poliedros unidos por un arco, o crear otros poliedros en el caso de caras colgantes. Esta fase requiere de varios criterios, ya que hay muchos casos borde que no se manejarían en caso contrario.

Bibliografía

[1] Alonso, Rodrigo: *A Delaunay Tessellation Based Void Finder Algorithm.* Tesis de Licenciatura, Universidad de Chile, Santiago de Chile, 2016.

[2] Balboa, Fernando, Pedro Rodriguez-Moreno y María Cecilia Rivara: *Terminal star operations algorithm for tetrahedral mesh improvement.* En *International Meshing Roundtable*, páginas 269–282. Springer, 2018.

[3] Barber, C. Bradford, David P. Dobkin y Hannu Huhdanpaa: *The Quickhull Algorithm for Convex Hulls.* ACM Trans. Math. Softw., 22(4):469–483, dec 1996, ISSN 0098-3500. https://doi.org/10.1145/235815.235821.

[4] Canepa, Aldo, Gonzalo Infante, Nancy Hitschfeld y Claudio Lobos: *Camarón: An Open-source Visualization Tool for the Quality Inspection of Polygonal and Polyhedral Meshes.* En *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 1: GRAPP, (VISIGRAPP 2016)*, páginas 130–137. INSTICC, SciTePress, 2016, ISBN 978-989-758-175-5.

[5] Guidotti, Valeria: *DELFIN++ Nueva versión de algoritmo de búsqueda de vacíos cosmológicos en 3D.* Tesis de Licenciatura, Universidad de Chile, 2021.

[6] Mengolini, Michael, Matías F. Benedetto y Alejandro M. Aragón: *An engineering perspective to the virtual element method and its interplay with the standard finite element method.* Computer Methods in Applied Mechanics and Engineering, 350:995–1023, 2019, ISSN 0045-7825. https://www.sciencedirect.com/science/article/pii/S0045782519301215.

[7] Salinas, Sergio, Nancy Hitschfeld, Alejandro Ortiz-Bernardin y Hang Si: *POLYLLA: Polygonal meshing algorithm based on terminal-edge regions.* CoRR, abs/2201.11925, 2022. https://arxiv.org/abs/2201.11925.

[8] Si, Hang: *TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator.* ACM Trans. Math. Softw., 41(2), feb 2015, ISSN 0098-3500. https://doi.org/10.1145/2629697.

[9] Sullivan, Bane y Alexander Kaszynski: *PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK).* Journal of Open Source Software, 4(37):1450, May 2019. https://doi.org/10.21105/joss.01450.

[10] Wriggers, Peter, Fadi Aldakheel y Blaž Hudobivnik: *Application of the Virtual Element Method in Mechanics.* páginas 4–5, Enero 2019.