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

ADAPTACIÓN Y EXTENSIÓN DE IMPLEMENTACIÓN DE UN ALGORITMO PARA GENERAR MALLAS POLIGONALES A OTROS LENGUAJES DE PROGRAMACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

BELTRÁN IGNACIO AMENÁBAR MEZA

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
VALENTIN MUÑOZ APABLAZA
FEDERICO OLMEDO BERÓN

SANTIAGO DE CHILE
2022

Resumen

Las mallas geométricas son una herramienta muy importante dentro de la computación gráfica. Estas permiten la representación de objetos complejos mediante el uso de elementos básicos como polígonos y con ello permiten realizar visualizaciones, simulaciones y resolver ecuaciones diferenciales de manera numérica. Es por esto último que ha surgido la necesidad de nuevos procesos de generación de mallas, no solo restringiéndose a mallas de triángulos, sino que permitiendo también mallas de polígonos.

A partir de esta problemática surge el algoritmo Polylla de generación de mallas de polígonos a partir de triangulaciones, junto con Polylla-Mesh, una implementación en C++ de este. Sin embargo, esta implementación es difícil de utilizar y deja afuera a científicos que usen otros lenguajes de programación como Python y MATLAB.

Es por esto que se decidió extender la implementación original, generando una versión más fácil de usar y también generando versiones para Python y MATLAB. Para esto, el código se refactorizó cambiando la estructura de datos a *Half-Edge*, actualizando el código a un estilo de C++ moderno y generando una librería que se pueda usar en C++.

Por otro lado, se utilizó la C++ MEX API de MATLAB para generar una interfaz de la librería a MATLAB y se utilizó pybind11 para generar una interfaz de la librería a Python, por lo que ambas interfaces son minimalistas y dependen de la versión de C++.

Los resultados obtenidos fueron positivos. La versión de C++ logró parecerse lo suficiente a la versión original, logrando que las mallas resultantes en promedio tengan un parecido de 96 % y logrando tiempos similares en las mismas triangulaciones dadas. Por otro lado, la interfaz de MATLAB funciona correctamente, permitiendo su fácil utilización, pero perdiendo un poco en eficiencia, logrando tiempos hasta 10 veces mayores en triangulaciones de muchos puntos en comparación a la versión de C++. Por ultimo la versión de Python funciona correctamente y también es fácil de utilizar, teniendo mejor rendimiento que la versión de MATLAB y acercándose a los tiempos de la versión de C++

A mi familia, polola y amigos, quienes siempre confiaron en mi y me apoyaron cuando lo necesitaba

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

Desde hace varias décadas, las mallas geométricas han ganado importancia, teniendo un rol importante como herramienta en diversas áreas, entre ellas la computación gráfica, sistemas CAD, métodos numéricos, sistemas de simulación de sólidos e incluso en astronomía.

Dentro del área de métodos numéricos, en conjunto con la física y la mecánica, se ha encontrado que las mallas poligonales son de gran utilidad como herramienta para sistemas de solución de ecuaciones diferenciales, existiendo métodos como el Método del Elemento Virtual (VEM) [7], que tienen menos restricciones sobre las mallas que las pedidas por otros métodos como el Método de los Elementos Finitos.

Con esto, surge la necesidad de tener algoritmos que permitan obtener mallas de polígonos en un dominio 2D, a partir de un conjunto de puntos o una malla ya establecida. Salinas, Si, Hitschfeld y Ortiz-Bernardin [19] desarrollaron el algoritmo Polylla que cumple estas características y puede generar una malla de polígonos a partir de una triangulación de Delaunay.

1.1. Motivación

En muchas áreas científicas se utiliza la programación como una herramienta de apoyo, dado que permite automatizar y facilitar tareas, realizar simulaciones, permitir visualizaciones y más. Sin embargo, muchas veces sucede que una tecnología o herramienta nueva requiere una gran cantidad de conocimiento sobre algún lenguaje de programación, *framework* o librería específico, haciendo que sea difícil de utilizar para la gente que no es del área.

En este caso se tiene un algoritmo generador de mallas poligonales que permite tomar una triangulación y generar una malla de polígonos, sin quitar puntos y solamente mediante la eliminación de aristas. Este generador podría ser utilizado y aprovechado por científicos del área de la física y mecánica por los descubrimientos que se mencionaron anteriormente para resolver ecuaciones diferenciales con métodos numéricos, pero dado el lenguaje en el que fue implementado, es difícil para los posibles usuarios utilizarlo, porque en general no lo manejan y en cambio utilizan o conocen otros lenguaje como Python y MATLAB.

Esta implementación, *Polylla-Mesh*, fue hecha en C++ a la par con el artículo del algoritmo, sin embargo, uno de los principales usuarios y parte de la misma investigación, Ortiz-Bernardin, trabajaba principalmente en MATLAB, por lo que el uso de *Polylla-Mesh* se tuvo que hacer mediante escritura y lectura de archivos para lograr interoperatibilidad entre *Polylla-Mesh* y el código que utilizaba las mallas con el método de elementos virtuales.

Además, el código original de *Polylla-Mesh* ni siquiera se puede usar a modo de librería en otros códigos, sino que solo se puede usar como un ejecutable en la línea de comandos. Esto genera la necesidad de modificar el código y agregar la posibilidad no solo de utilizar este generador de mallas como una librería en C++, sino también llevarlo a más usuarios, generando versiones en otros lenguajes.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo general de esta memoria es adaptar y extender la implementación de un algoritmo de generación de mallas poligonales, para permitir su uso como una librería desde C++ y generar una interfaz de esta para su uso desde otros lenguajes de programación. El código final debe poder usarse como una librería desde C++, Python y Matlab. Por último, el código debe ser extensible para permitir la creación de nuevas interfaces a otros lenguajes de programación o la integración con librerías de geometría computacional.

1.2.2. Objetivos Específicos

- Generar una pequeña librería a partir del código del algoritmo, que sea extensible y flexible
- Generar una interfaz para poder utilizar la librería en Python
- Generar una interfaz para poder utilizar la librería en MATLAB
- Generar documentación del código y una guía de uso

1.3. Descripción de la solución

La solución consiste principalmente en generar una librería de C++ a partir del código ya existente y con ello generar interfaces a cada uno de los lenguajes, es decir, código que permita utilizar el algoritmo en ese lenguaje, pero que por debajo llamen al código de C++ de la implementación original, logrando así facilidad de uso y también lograr la eficiencia que tenia el código original.

Para esto se utilizó la *C++ MEX API* en el caso de MATLAB y pybind11 en el caso Python, ambas elegidas de un grupo de varias opciones para realizar las interfaces, de manera de reducir la complejidad de ambos códigos y facilitar su implementación, mantención y uso.

1.4. Estructura de la memoria

La estructura de los capítulos siguientes de esta memoria es la siguiente:

En el capítulo 2 se plantean los conceptos necesarios para comprender esta memoria, junto con el estado del arte. En el capítulo 3 se hace una descripción a fondo del problema a resolver, mostrando detalles sobre el código original y discutiendo los requisitos que debería tener la solución.

En el capítulo 4 se desarrolla la solución, mostrando el diseño de esta y como se llegó a obtenerla. Posteriormente, en el capítulo 5 se explica en detalle como se realizó la comparación entre la solución obtenida y el código original, para luego mostrar los resultados de aquella comparación.

Por ultimo, en el capítulo 6, se analiza y concluye a partir de los resultados y lo expuesto en los capítulos anteriores, planteando también mejoras que se le podrían hacer y posible trabajo futuro relacionado.

Capítulo 2

Antecedentes

2.1. Conceptos básicos

Antes de introducir al problema principal es necesario definir algunos conceptos que son relevantes a lo largo de la memoria.

2.1.1. Mallas Geométricas

Una malla geométrica es una colección de varios polígonos adyacentes en un espacio. Esta puede ser descrita por sus elementos básicos, que son vértices, aristas y caras. Generalmente son usadas para representar objetos geométricos más complejos, como superficies u objetos 3D, puesto que son más sencillas y permiten descomponer un objeto en varias partes. Es por esto mismo, que son un elemento base de la computación gráfica, siendo muy utilizadas en *rendering* 2D, 3D y sistemas CAD. Es importante notar que, a pesar de que la definición que se acaba de dar solamente considera mallas 2D, las mallas geométricas pueden ser en más dimensiones, existiendo mallas de poliedros y de politopos de más dimensiones.

En computación existen varias formas de representar una malla geométrica. La forma más clásica es basada en caras, donde se guarda la información de los vértices y la información sobre que vértices conforman la cara. Esto permite no guardar la información sobre las aristas, guardándolas implícitamente en la cara.

También existen las representaciones basadas en aristas. Dentro de estas existe *Quad-Edge* [11] y *Winged-Edge*. Todas estas enfocan el guardado de información en las aristas, dejando que las caras sean implícitas o que tengan la menor información posible.

Por último, existe la estructura de datos *Half-Edge* [13], también conocida por *doubly connected edge list* o *DCEL*. Esta estructura de datos divide cada arista en dos partes, cada una con una cierta orientación. Cada uno de los *half-edges* tiene un vértice de inicio, junto con un *half-edge* siguiente, uno anterior y un gemelo. A veces también se guarda la única cara a la cual está asociado el *half-edge*. En la figura 2.1 se puede ver un ejemplo de una malla

que usa *half-edge* y como se relacionan los elementos de este tipo de estructura de datos.

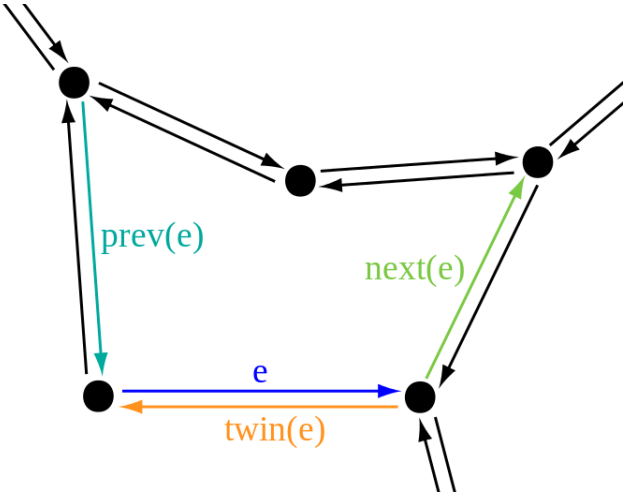

Figura 2.1: Ejemplo de estructura de *half-edge*. Fuente: Wikimedia [1]

Esta estructura es más compleja que las otras, pero facilita el movimiento dentro de una malla, por ejemplo logrando que sea sencillo recorrer alrededor de una cara asociada. Todo esto se debe a que es una estructura que se enfoca principalmente en vecindades.

2.1.2. Mallas de triángulos

Uno de los tipos de mallas más utilizados son las mallas de triángulos, dado que el triángulo es el polígono más sencillo que se puede tener, siendo la figura geométrica más sencilla para definir planos. Es por esto que existen varias formas de generar mallas de triángulos, las que comúnmente se llama triangulaciones.

Una de las triangulaciones más conocidas es la de Delaunay [8]. El algoritmo más usado para generar esta triangulación se basa en insertar puntos incrementalmente (también existe una versión *divide and conquer* que tiene mejor rendimiento) intentando mantener la propiedad de Delaunay. Esta propiedad consiste en que para cada triángulo, ningún vértice de otro triángulo debe estar dentro de su circuncírculo.

Por cada par de triángulos adyacentes que no cumplan la condición se debe generar un *flip* de la arista que los conecta, generando un par de nuevos triángulos en su lugar. Esto se debe realizar recursivamente por cada nuevo triángulo en la malla, corrigiéndola hasta que no hayan triángulos adyacentes que no cumplan con la condición de Delaunay. La operación de *flip* se puede ver mejor en la figura 2.2, donde el par de triángulos inicial ABD y BCD no cumplen la condición de Delaunay, teniendo al vértice A dentro del circuncírculo del triángulo BCD (en azul). Para solucionarlo, se realiza el *flip*, cambiando la arista DB por AC y generando los triángulos ABC y ACD. Esto último genera que se cumpla la condición de Delaunay, dado que se tiene al vértice D fuera del circuncírculo de ABC (en rojo).

Es importante saber además que esta triangulación garantiza que el ángulo mínimo se vea maximizado, por lo que es la triangulación lo más “equilatera” posible.

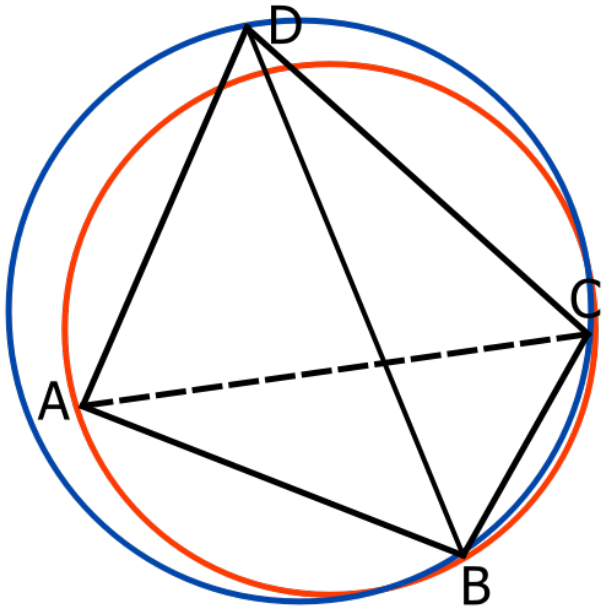

Figura 2.2: Ejemplo de *flip* de aristas para garantizar la condición de Delaunay

Además de triangulaciones, existen procedimientos para reducir o aumentar el detalle de una malla (la cantidad de triángulos en cierta área de la malla), como también de mejorarla. Dentro de los refinamientos, existe un algoritmo desarrollado por Rivara [16] basado en caminos generados por sucesiones de triángulos conectados por sus lados más largos. Este concepto se conoce como *LEPP* o *Longest Edge Propagation Path* y consiste en tomar un triángulo cuyos ángulos pequeños sean menores que lo deseado. Se realiza la búsqueda de su LEPP correspondiente y se hace la inserción de un punto en la arista terminal del LEPP, conocida como *terminal edge*. Se debe proceder igual que en la triangulación de Delaunay, produciendo que los triángulos sean reemplazados por otros triángulos más refinados que cumplan las condiciones del ángulo mínimo.

2.1.3. Mallas de polígonos

En relación a mallas de polígonos que no sean necesariamente triángulos, existen varias formas de generarlas, siendo muy conocidos los diagramas de Voronoi. Estos consisten en particionar el plano en el que están los puntos iniciales, generando regiones poligonales convexas tal que cada punto dentro de esta región debe estar más cerca de algún punto inicial que de todos los demás. Esto genera que cada arista de los polígonos generados esté a la misma distancia de los puntos que separa. En la figura 2.3 se puede apreciar un diagrama de Voronoi con los puntos iniciales.

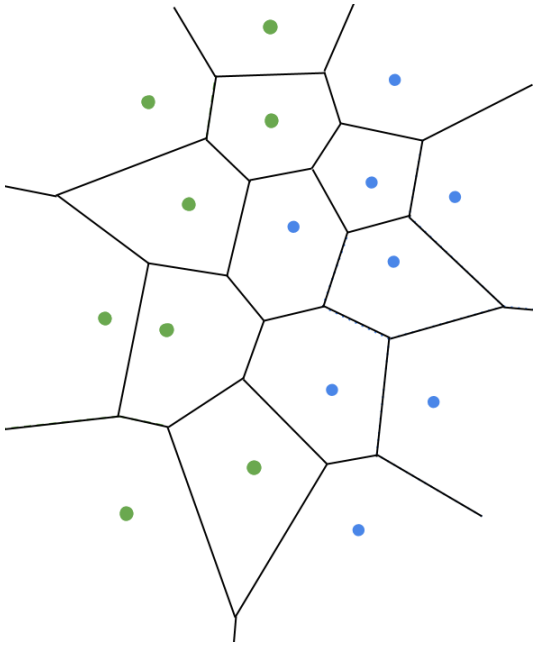

Figura 2.3: Diagrama de Voronoi. Fuente: Wikimedia [25]

Es importante mencionar que existe una relación entre un diagrama de Voronoi de un conjunto de puntos y su triangulación de Delaunay. Dado que el diagrama de Voronoi maximiza la distancia entre los puntos y las aristas, los vértices generados de la intersección de las aristas están a la misma distancia de los puntos iniciales. Esto genera que un vértice del diagrama de Voronoi sea el circuncentro del triángulo generado por los 3 puntos en las regiones que se generan de la intersección de las aristas. Con esto, si se tiene una triangulación de Delaunay de un conjunto de puntos, se pueden calcular los circuncentros de los triángulos y a partir de esto generar el diagrama de Voronoi conectando los circuncentros de triángulos adyacentes. Esto se puede apreciar en la siguiente figura:

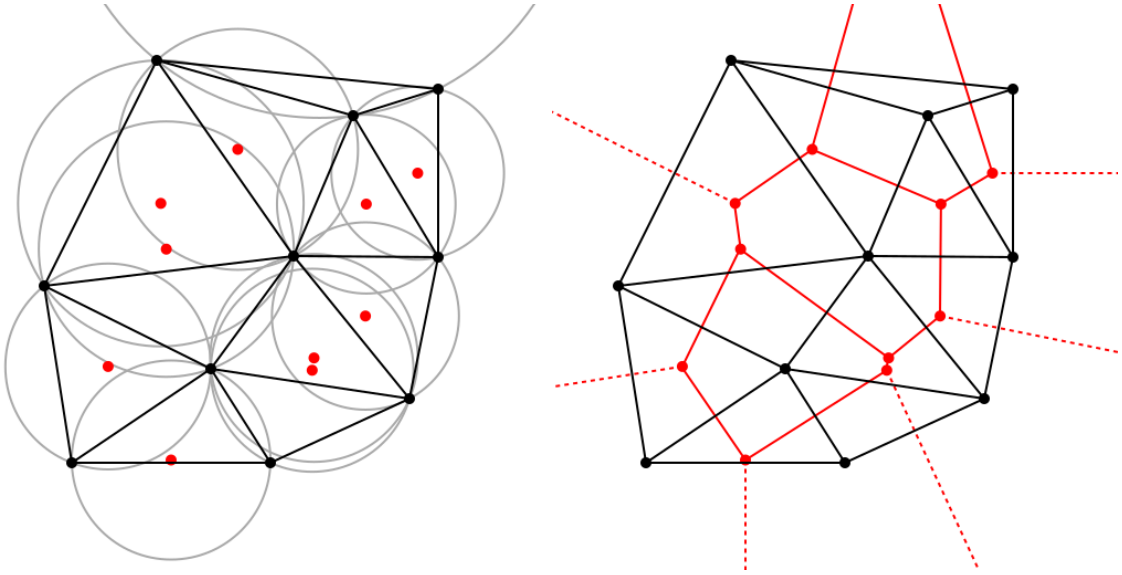

Figura 2.4: Relación entre triangulación de Delaunay (izquierda) y diagrama de Voronoi (derecha) de un conjunto de puntos. Fuentes: Wikimedia [10][9]

Los diagramas de Voronoi se realizan particionando todo el espacio, por lo que genera regiones poligonales convexas, pero también existen regiones infinitas que no nos sirven como parte de una malla resultante. Es por esto que se utiliza la versión restringida de los diagramas, en la que se marcan aristas que corresponden al borde de la figura y que deben respetarse, logrando así una malla que sea cerrada. También es importante notar que los polígonos generados no contienen a los puntos originales que se usaron para su generación, por lo que a pesar de que si genera una malla de polígonos, esta malla no contiene los puntos iniciales dados.

Dado lo anterior mencionado, Salinas et al. [19], desarrollaron un algoritmo que toma un conjunto de puntos, genera una triangulación de Delaunay y puede generar una malla de polígonos que si contiene esos puntos. En la figura 2.5 se muestra el tipo de mallas que genera. Se puede apreciar que a diferencia de Voronoi, permite crear polígonos no convexos.

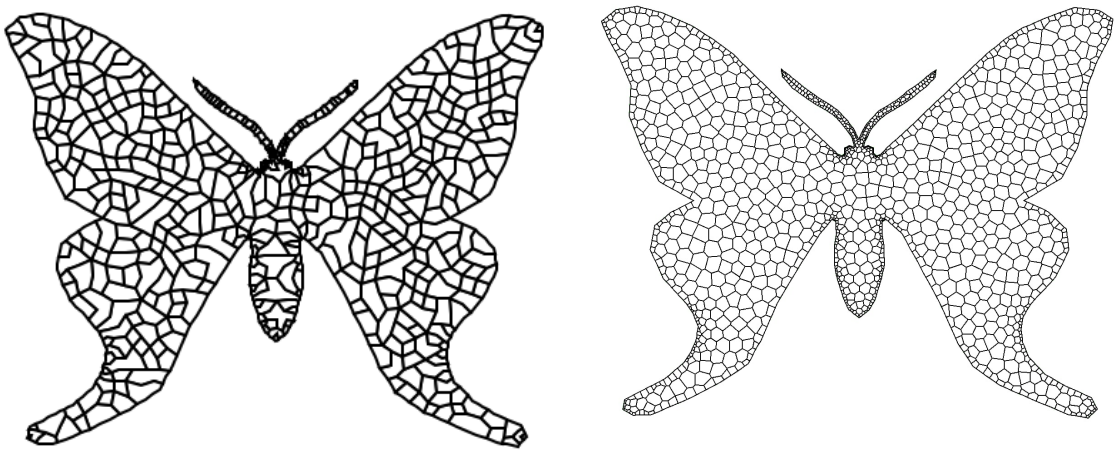

Figura 2.5: Mallas de polígonos generadas a partir del algoritmo de Polylla (izquierda) y del diagrama de Voronoi restringido (derecha) usando la misma triangulación inicial

La base del algoritmo es generar los polígonos a partir de unión de triángulos, quitando aristas. Para esto se apoya en el concepto anteriormente explicado de LEPP, buscando particionar la triangulación en regiones que compartan *terminal edges*. El algoritmo busca por cada triángulo su *LEPP* y mediante esto lo relaciona a uno de los *terminal edges* y por ende a una de las regiones disjuntas, en adelante *Terminal-edge region* [15]. Sin embargo, realizarlo de esa manera es poco conveniente y eficiente, por lo que a continuación se describe el algoritmo en sí.

El algoritmo se divide en 3 partes. En primer lugar se tiene la fase de etiquetado o *label phase*, en donde se identifican todas las aristas de la malla, basándose en si son la arista más grande de algún triángulo; la más grande de ambos, o ninguno de los dos. Esto equivale a la búsqueda de los *LEPP* de cada triángulo, pero pensando en la construcción directa del polígono, por lo que si una arista no es la más grande de ningún triángulo, va a corresponder a una arista del borde de la *terminal-edge region*, la cual se denomina *frontier-edge*. Cuando se tiene el caso donde la arista es mayor de uno de los triángulos, corresponde a una arista que va a quedar en el interior de la *terminal-edge region*, por lo que se le denomina *longest-edge*

o *interior-edge*. Por último, en el caso donde la arista comparte ambos triángulos, se tiene una *terminal-edge*.

En segundo lugar se tiene la fase de construcción del polígono o *traversal phase*. Esta fase consiste en recorrer una *Terminal-Edge Region*, usando un triángulo como base o semilla y moviéndose a partir de este por triángulos adyacentes, con la intención de ir encontrando el borde del polígono a armar. Una vez recorrida la *Terminal-Edge Region*, con el borde completamente definido se tiene un candidato a polígono. Esto quiere decir que recorriendo las *frontier-edges* se puede establecer el candidato a polígono.

Notar que la fase anterior genera solamente candidatos a polígonos, dado que es posible que una de las *frontier-edge* no esté en la cobertura de la *terminal-edge region*, sino en su interior. Estas aristas se denominan *barrier-edges* y crean lo que se denomina un *barrier-edge tip*, es decir, un vértice completamente aislado de cualquier otra *frontier-edge*. Esto genera que el polígono resultante de la *traversal phase* no sea un polígono simple (ver figura 2.6), lo cual motiva la siguiente fase.

Por último se tiene la fase de reparación de polígonos no simples, también llamada *reparation phase*. Esta fase, arregla el problema de los *barrier-edge tips* extendiendo las aristas problemáticas y dividiendo las figuras candidatas a polígonos en 2 figuras más pequeñas por cada uno de estos casos problemáticos, generando polígonos más pequeños que si son simples.

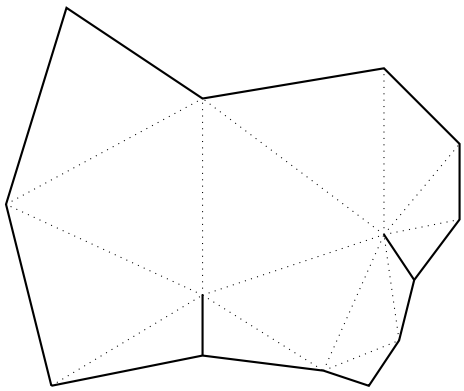

Figura 2.6: Ejemplo de figura que no constituye un polígono simple, presentando aristas en su interior

2.2. Estado del arte

En esta sección se va a hablar del estado del arte de 3 temas importantes en esta memoria. El primero trata sobre herramientas ya existentes que permitan hacer lo mismo que las implementaciones a hacer. El segundo tema se refiere al uso actual de C++, lenguaje de programación utilizado en la mayor parte del código generado en este trabajo de título. Por último, el tercer tema se refiere a el uso de interfaces entre lenguajes, utilizando a MATLAB y Python como principales ejemplos, dado que estos van a ser los lenguajes objetivos para las implementaciones a realizar.

2.2.1. Herramientas generadoras de mallas de polígonos

Como se mencionó anteriormente existe una gran cantidad de algoritmos que permiten generar mallas, sin embargo, es difícil encontrar un generador de mallas que se centre en mallas de polígonos cualquiera. En esta sección se mostraran varias herramientas disponibles que se pueden usar para generar mallas de triángulos.

En primer lugar se tiene a la forma más común de generar mallas de polígonos, que corresponde a los diagramas de Voronoi. Todas las librerías de geometría computacional tienen una versión del algoritmo que genera diagramas de Voronoi a partir de puntos. En el caso de C++, CGAL tiene clases y funciones dedicadas para esto; en Python, la librería Scipy tiene también una función para eso; la herramienta qhull también tiene dentro de sus herramientas un generador de diagramas de Voronoi y MATLAB también posee una funcionalidad para eso. El único problema, es que estos diagramas de Voronoi no son restringidos, por lo que no se generan las mallas necesitadas y se deben modificar manualmente para lograr lo requerido. A pesar de lo anterior, existen generadores de diagramas de Voronoi que si permiten la versión restringida, como el caso de Detri2 [21].

Luego de esto, tenemos un generador de mallas de polígonos que se basa en el diagrama de Voronoi restringido y mejora la malla generada moviendo los puntos hacia los centroides calculados en funciones de los polígonos vecinos. El algoritmo, presentado por Talischi et al. [24] genera mallas de polígonos muy cercanos a regulares, teniendo una implementación muy pequeña en MATLAB de 134 lineas.

Por otro lado, tanto Ooi et al. [14] y Song [22] presentan formas de generar mallas de polígonos basadas en la utilización de Voronoi restringido, generando los puntos a utilizar desde una descripción del dominio necesitado, pero no existe ninguna implementación al respecto, por lo que no es posible utilizarlas sin implementarlas desde 0.

Como se puede ver todas las formas de generar mallas de polígonos anteriores utilizan Voronoi y por ende los puntos iniciales que describen un dominio son prácticamente descartados cuando se genera la malla final.

Por último, Salinas et al. [19] presentan el algoritmo ya explicado en 2.1.3, que genera mallas de polígonos a partir de una triangulación o un conjunto de puntos. Este algoritmo tiene una implementación en C++ disponible en un repositorio en Github [18].

2.2.2. Uso de C++ en software actual

C++ es un lenguaje de programación que se utiliza en una gran cantidad de software actual. Introducido por primera vez en 1985, fue pensado como una extensión de C agregando el uso de clases para poder utilizar programación orientada a objetos, por lo que fue pensado como un lenguaje de bajo nivel, con una gran flexibilidad, pero enfocado en el rendimiento.

Con el paso del tiempo, en cada nueva versión de C++ se han ido agregando nuevas características y funcionalidades al lenguaje, según lo que el estandar ISO va dictando, por lo que C++ ha dejado de ser un lenguaje solamente enfocado a programación orientada a

objetos y ahora es multiparadigma, teniendo cada vez más herramientas para desarrollar cualquier tipo de software.

Por lo anterior es que se hace la diferencia entre el uso de C++ en sus versiones antiguas y las nuevas, estableciendo un estilo de *C++ moderno*. Este estilo comunmente se refiere al uso de las caracteristicas que provee el lenguaje en el estandar más nuevo (a la fecha de este trabajo C++20), usualmente utilizando caracteristicas agregadas desde C++11 en adelante, versión en la que se agregaron contenedores nuevos (`vector`, `array`, `tuple`), *move semantics* y *r-value references*, *smart pointers*, *range-based for loops* y otras herramientas como `chrono` y `async`. Además, en versiones posteriores se agregaron lambdas, tipos dinámicos, mejoras en el sistema de paralelismo, entre otros. Es importante notar que manejar y utilizar C++ moderno es cada vez más relevante para el desarrollo de software, dado que provee mayor robustez y flexibilidad, minimizando errores y mejorando el rendimiento.

2.2.3. Uso de interfaces a otros lenguajes

Uno de los problemas más grandes hoy en día es que existen tantos lenguajes de programación que cuando se realiza una librería grande en cierto lenguaje, inmediatamente se crea la necesidad de que pueda ser utilizada en otro. Esto hace surgir la principal pregunta para los mantenedores y desarrolladores de la librería: ¿Cual es la mejor forma de portar el código o generar una implementación en otro lenguaje?

La primera forma es tomar la descripción del algoritmo y programarlo desde 0 utilizando las herramientas que provee el lenguaje al cual se quiere portar. Esto es bastante caro de realizar tanto en tiempo como en desarrollo y tiene problemas asociados como la dificultad de mantener más de una versión del código, para que tenga las mismas *features*. Esto se suma a que muchas veces el código original está hecho en un lenguaje de bajo nivel, con una gran cantidad de optimizaciones, por lo que es complejo obtener el mismo rendimiento en una versión de un lenguaje de más alto nivel.

Por otro lado, existe una forma que es bastante popular hoy en día y que consiste en llamar al código original desde el lenguaje al cual se quiere portar y generar solo una pequeña interfaz que se encargue de la conversión entre estructuras de datos nativas del lenguaje al que se porta y las estructuras de datos usadas en la librería original.

Podemos ver que lo segundo se puede hacer en Python, donde se puede ver código hecho de esta manera en grandes librerías como *numpy*, *scipy*, *pandas* y *pytorch*. Para esto se puede usar la librería oficial que provee CPython, que posee el *header* `Python.h` y que permite la creación de una extensión del lenguaje mediante código de C o C++. Además de la manera oficial existen varios otras opciones que se muestran a continuación:

- Python.h: La forma nativa de generar extensiones al lenguaje, está pensada para ser usada en el lenguaje C
- Ctypes: Una forma de cargar desde Python archivos de librería dinámicos `.dll` o `.so`, de manera de que se puede llamar funciones de código de C o C++ directamente.

- PyBind11: Una librería de C++ que se encarga de simplificar la conversión de C++ a Python. Soporta el estándar de C++11 y es una librería *header only*.
- Boost.Python: Un paquete de la librería Boost dedicado a la integración de C++ con Python.
- cppyy: Una librería de Python para convertir en runtime código de C++ a Python. La conversión es automática y permite incluso usar librerías externas u otras formas de integrar el código.
- Swig: Una herramienta y lenguaje que permite convertir programas de C++ a otros lenguajes de mayor nivel, el lenguaje de Swig permite especificar en un *interface file* exactamente que componentes se quieren exponer a la interfaz y cómo deben hacerlo.

Por el lado de MATLAB, existe una forma oficial también de portar código, tanto de FORTRAN, como de C y C++. Para esto se ofrecen varias opciones, siendo la principal una herramienta llamada `MEX` que permite crear funciones en C y C++ (en adelante funciones MEX), que al ser compiladas a binarios pueden ser llamadas desde MATLAB.

La herramienta MEX provee 2 APIs principales, está la versión de C, llamada *C MEX API* y que permite utilizar estructuras de MATLAB mediante la *C Matrix API* y por otro lado está la versión de C++, llamada *C++ MEX API*, que se puede utilizar junto a la *Data API for C++* que provee las estructuras de datos de MATLAB y también acceso al *engine* de MATLAB en si mismo.

La otra opción existente en MATLAB corresponde a la herramienta `clibgen`, que permite convertir librerías de C o C++ a un formato llamado *MATLAB® Live Code definition file*, el cual permite crear interfaces usables en MATLAB. Sin embargo, el soporte que tiene para C++ moderno es bajo y la definición de la librería en el formato que requiere MATLAB es bastante cerrado y difícil de utilizar.

Capítulo 3

Problema

Como fue mencionado anteriormente, dentro del área de métodos numéricos y optimización se tiene constantemente la necesidad de trabajar y resolver problemas en dominios complejos usando mallas. Para el método de elementos finitos es suficiente trabajar con mallas de triángulos y cuadriláteros, pero esto requiere condiciones específicas sobre ángulos que no siempre se tienen o que requieren de refinado extra de las mallas para su uso. Por otro lado, métodos como el del elemento virtual dan mayores libertades con respecto al tipo de malla, permitiendo la utilización de mallas de polígonos.

Con estos nuevos métodos, surgió la necesidad de tener nuevas formas de generar mallas geométricas a partir de la descripción de un dominio, por lo que que surgieron varios algoritmos mencionados en 2.2.1. A pesar de lo anterior, encontrar librerías o implementaciones con aquellos algoritmos es bastante difícil, dado que los artículos solo describen los algoritmos en pseudocódigo, muestran código minimalista o las librerías no contienen documentación al respecto y son complicadas de instalar y usar.

3.1. Polylla-Mesh

En el caso del algoritmo Polylla-Mesh se tiene algo parecido a los otros generadores de mallas de polígonos. La implementación fue hecha a la par de la investigación y también es descrita en pseudocódigo en el artículo, sin embargo, tiene escasa documentación y solo puede ser usada como un ejecutable. Lo anterior tiene como consecuencia que como entrada y salida solamente se pueden usar archivos, dejando de lado la posibilidad de ser usada como librería e integrada en otros códigos. Por otro lado, a pesar de estar escrita en C++, tiene una gran parte del código utilizando funciones de librería y el estilo de programación de C, lo que genera mayor dificultad para entender el código.

Lo anterior hace que Polylla-Mesh sea igual de complejo de usar que las otras opciones, sumado a que los posibles usuarios de este algoritmo (fuera del área de mallas geométricas) en general tienen mayor conocimiento de otros lenguajes como MATLAB y Python, por lo que sería una barrera de entrada más al uso de esta implementación.

Es por esto que surge la necesidad de tomar Polylla-Mesh y modificarlo para facilitar su uso a un mayor número de usuarios, haciendo que sea más fácil de usar, que se pueda usar en otros lenguajes de programación en donde se pueda necesitar y también que tenga una documentación disponible.

3.2. Código original

Para poder identificar completamente el problema mencionado, primero es necesario hacer un análisis más detallado del código de Polylla.

La implementación original consiste en un archivo `main.cpp` principal que contiene la implementación del algoritmo y utiliza funciones auxiliares que están implementadas en otros archivos, a continuación se detalla cada uno de los archivos existentes.

- `main.cpp`: El archivo principal. Contiene principalmente una función *main* que lee desde `argv` los archivos de input y los nombres de los archivos de output. Ejecuta las 3 partes del algoritmo y realiza las mediciones de tiempo necesarias.
- `BET_elimination.cpp/.h`: Contiene funciones relacionadas a la fase de reparación de los polígonos, que sirven para corregir *barrier-edge tips* de polígonos no simples.
- `consts.h`: Un *header* simple con constantes que se usan en varias partes del código. Estas constantes están definidas mediante la macro `#define`.
- `delaunay.cpp/.h`: No necesitados por el proyecto. Anteriormente utilizados cuando el código necesitaba tener la librería detri2 para generar la triangulación.
- `hashtable.cpp/.h`: Define la estructura `node`, junto con varias funciones para crear una *hash table*. Se usan en la *reparation phase* del algoritmo.
- `io.cpp/.h`: Contiene las funciones para leer y escribir los archivos con formatos correspondientes.
- `mesh.cpp/.h`: Implementa 2 funciones para trabajar con la malla final obtenida.
- `metrics.cpp/.h`: Contiene una función principal que se encarga de calcular las métricas sobre mallas y también varias funciones auxiliares relacionadas al mismo proceso.
- `polygon.cpp/.h`: Contiene funciones para generar los polígonos a partir de las seed edges, junto con funciones para trabajar con estos polígonos.
- `SmallestEnclosingCircle.cpp/.h`: Una pequeña librería que se usa para calcular el circulo más pequeño que contiene a los polígonos resultantes de la malla, usado principalmente para calcular métricas.
- `triang.cpp/.h`: Una gran variedad de funciones para trabajar con triángulos y las triangulaciones.

Las estructuras de datos que se utilizan son principalmente arreglos, teniendo también estructuras auxiliares, como la de `node`, cuyo fin es ser usado en una *hashtable* y las estructuras de la librería *Smallest enclosing circle*, que son `Point` y `Circle` y que no son usadas en el código principal, sino que únicamente para realizar el cálculo de métricas.

3.3. Requerimientos

Para generar una solución al problema planteado anteriormente se necesita que la solución cumpla una serie de requisitos especificados a continuación:

En primer lugar, es necesario convertir la implementación en una librería, de manera de que no sea solamente utilizable como un ejecutable, sino que también pueda ser usado en otros proyectos sin la necesidad de ser ejecutado externamente. Esto implica de que el código se debe independizar de la necesidad del uso de archivos con formatos específicos y dejando eso último solamente al ejecutable.

En segundo lugar, la librería generada debe ser utilizable desde otros lenguajes de programación además de C++, específicamente Python y MATLAB, teniendo todas las funcionalidades de la versión original y conservando la eficiencia del código original.

En tercer lugar, es importante de que sea utilizable en una gran cantidad de dispositivos, por lo cual debe haber soporte multiplataforma. El código original soporta esto mediante *CMake*, por lo que se debe conservar ese sistema de compilación, mientras que para el caso de Python y MATLAB, dada su naturaleza multiplataforma, debe darse el mismo soporte para las librerías generadas.

Por último, dada la escasez de documentación en el código original, es necesario que todo el código generado esté bien documentado y tenga información de como compilarse, instalarse y usarse.

Capítulo 4

Diseño e implementación de la solución

Como se había mencionado anteriormente, para lograr que una mayor cantidad de personas tenga acceso a la implementación de Polylla y poder usar el algoritmo, se decidió generar versiones del algoritmo en Python y MATLAB, dado que el primer lenguaje es de los lenguajes mas usados hoy en día y tiene una gran cantidad de librerías y uso en el área científica. Por otro lado, se eligió MATLAB dado que las simulaciones usadas en la investigación original utilizaban las mallas generadas en simulaciones hechas en este lenguaje, por lo que generar una librería fácil de usar e integrar para su utilización en MATLAB aportaría positivamente en este área.

Para las dos interfaces/librerías está la posibilidad de portar el código desde 0, sin embargo, dada la naturaleza de estos lenguajes, es probable que se pierda eficiencia y no tenga el mismo rendimiento de la implementación original. Por esto mismo, la idea es usar las herramientas que provee cada lenguaje y llamar el código original por debajo.

Esto permite que la mantención del código a futuro sea más sencilla, dado que en vez de mantener 3 códigos distintos y realizar los mismos cambios a los 3 códigos, solamente se necesite cambiar la implementación original y el llamado que se hace desde las interfaces.

4.1. Polylla como librería

Para lograr el primer objetivo planteado, se partió realizando una revisión del código original, entendiendo su funcionamiento y utilizándolo con varias triangulaciones, realizando el mismo análisis planteado en 3.2.

La idea inicial era tomar ese código, modificarlo y con ello generar una nueva versión para que funcionara como librería. Esto no fue necesario, dado que al mismo tiempo que se empezó a trabajar en esta memoria, se estaba realizando una reimplementación del algoritmo, utilizando C++ moderno y cambiando la estructura de datos principal usada para la triangulación por una basada en *half-edge*, en vez de utilizar solo arreglos como en la original.

4.1.1. Implementación de C++ moderno

Esta versión, en adelante *Polylla-Mesh-DCEL*, tiene varias mejoras en comparación con la anterior, partiendo por su menor tamaño, dado que al utilizar la sintaxis de C++, el manejo de *input/output* es más sencillo (mediante el uso de *streams*), se pueden usar *range-based for loops* y se puede aprovechar un enfoque orientado a objetos.

Con respecto a la estructura del proyecto, este solamente contiene 3 archivos de código: `main.cpp`, `polylla.hpp` y `triangulation.hpp`, donde los últimos dos corresponden a una librería *header only* y el primero un archivo con una función *main* para generar el ejecutable. Además posee archivos para compilar con CMake, al igual que la versión anterior, y esto permite la compilación en multiplataforma.

Como se mencionó anteriormente, esta implementación tiene como objetivo utilizar una representación de *half-edge* para las triangulaciones, por lo que la mayoría de las estructuras de datos son cambiadas, definiendo las siguientes clases:

- `vertex`: Representa un punto con sus dos coordenadas. También contiene información adicional sobre si están en un borde y a que arista corresponden.
- `halfEdge`: Representa a un *half-edge*. Posee información de los dos puntos a los que está conectado, el *half-edge* siguiente, anterior y opuesto, como también la cara de la que es parte y si es parte de un borde. Notar que todos estos datos se guardan en forma de índices.
- `Polygon`: Representa un polígono resultante del algoritmo. Tiene una lista de vértices que lo conforman y la arista semilla que lo genera.
- `Triangulation`: Representa una triangulación inicial. Tiene la información sobre los vértices, triángulos y luego *half-edges* que la conforman. Además contiene métodos que permiten moverse dentro de la malla usando los *half-edges*.
- `Polylla`: Es la clase principal que maneja el algoritmo. Guarda la triangulación, la lista de polígonos generados y vectores de bits que permiten marcar condiciones y guardar más información sobre cada arista de la triangulación. Los métodos de esta clase son mayoritariamente relacionados al algoritmo de generación de mallas de polígonos.

Una de las características importantes de estas estructuras, es que están pensadas para ser usadas en conjunto, dado que usan índices en vez de referencias o punteros a otros elementos. En el caso de la estructura de `half-edge`, es necesario tener una lista de puntos y de `half-edge` para tener índices, de lo cual se encarga `Triangulation`, encargándose incluso de realizar los cálculos con respecto a las operaciones clásicas para moverse al siguiente, anterior, opuesto y operaciones más complejas como rotar en sentido reloj con respecto a un vértice.

El flujo del código tiene la misma estructura que la versión anterior, siendo completamente dependiente de archivos. Esto se puede ver en que los parámetros del constructor de `Triangulation` son los nombres de los archivos que contienen los datos sobre la triangulación

a utilizar. A pesar de eso, ya se cumple uno de los objetivos, dado que es posible importar Polylla como librería en otro código de C++, por la forma en la que está estructurada el código.

4.1.2. Cambios realizados

Polylla-Mesh-DCEL fue tomado y se le realizaron varios cambios para lograr a cabalidad los objetivos planteados.

En primer lugar el constructor de `Triangulation` crea y carga la triangulación desde archivos, por lo que los parámetros de este son *strings* con los nombres de los archivos. Esto se cambió, agregándole nuevos constructores a partir de objetos de C++, principalmente a partir de una lista de puntos y una lista de índices que conforman los triángulos. Notar que no se dejaron de lado los constructores a partir de archivos, dado que pueden servir para seguir siendo compatible con el ejecutable que funciona con archivos.

El constructor de `Polylla` no recibía una triangulación, sino que los parámetros eran pasados directamente al constructor de `Polylla`, para que dentro de este mismo se creara la triangulación. Esto agregaba la existencia de una dependencia innecesaria entre `Polylla` y `Triangulation`, y hacía que `Polylla` ahora también dependiera directamente de archivos. Para arreglar esto, se cambiaron los constructores de Polylla a un único constructor que tiene como parámetros un objeto de la clase `Triangulation`. De esta manera, el usuario tiene que crear la triangulación por si mismo y luego pasarla a `Polylla`, quitando la dependencias innecesarias.

Lo siguiente que se realizó fue establecer una guía de estilo para el código, mediante el uso de *clang-format*, de manera de que se mantenga un mismo formato en todo el código. Esto es importante, dado que el código no tenia consistencia al respecto y dificultaba la compresión de este. Principalmente se renombraron nombres de clases, métodos y variables; se estableció una regla para el espaciado del código, incluyendo indentación; se agregaron reglas sobre el posicionamiento de paréntesis, llaves, asteriscos y más; se estableció un tamaño máximo de caracteres por linea, y reglas para los comentarios.

Luego de esto se arreglaron los archivos de CMake relacionados con la compilación, dado que en algunos sistemas habían errores relacionados a que Polylla era una librería *header only* y se estaban buscando los archivos de librería para compilarlos con el ejecutable de Polylla. Esto sucedía porque la librería se había establecido como una librería estática en vez de usar la opción *header only* que da CMake.

Se reorganizaron los archivos, dejando cada estructura o clase en un archivo aparte. De esta manera es posible importar específicamente cada una, sin importar cosas extra que no se van a utilizar.

Se activaron todos los *warnings* en la compilación y se arreglaron todos los que aparecían al compilarlo, aumentando la robustez del código frente a posibles errores por overflow y precisión al utilizar `int` y `size_t`. Junto con esto, se aprovechó de agregar *const correctness*, referencias, *range-based for loops* donde era necesario y cambiando `typedef` por `using`. Todo

esto le agrega robustez al código, haciéndolo menos propenso a errores y acercándolo al estilo de programación de C++ moderno.

Todos estos cambios generan que ahora el código correspondiente al ejecutable de Polylla (principalmente el código en `main.cpp`) tenga un par de cambios, teniendo que construir primero el objeto de la triangulación y luego con eso construir Polylla como tal, para realizar la generación de la malla a partir de la triangulación. Esto genera que sea un par de lineas más complejo, pero nos permite que ahora Polylla pueda ser utilizada desde C++ sin la necesidad de utilizar archivos, por ejemplo generando la triangulación con detri2 (como en algún punto se usó en Polylla-Mesh), usando CGAL u otra librería similar.

4.2. MATLAB

Como se mencionó en la sección 2.2.3 en MATLAB existen varias formas de extender el lenguaje con código de C++ y C. Para elegir la opción a usar se consideró como objetivo que la interfaz debía ser lo más sencilla posible, tener un buen soporte para C++ moderno y que la librería debía ser fácil de instalar (o compilar) y usar.

De las 3 opciones disponibles se eligió utilizar la *C++ MEX API*, junto con la *Data API for C++*. Esto se debe a que esta opción cumple con los requisitos mencionados anteriormente, teniendo una excelente compatibilidad con C++ moderno, la cantidad de código necesario para generar una función es baja y genera un binario que es importable directamente en MATLAB, por lo que su instalación y uso es muy sencillo.

Por otro lado, a pesar de que la segunda API que provee MATLAB, *C MEX API* junto con *C Matrix API*, también generan binarios fáciles de usar y tampoco se necesita mucho código para generar funciones, no facilita el uso de C++ moderno y se tendrían que hacer cambios en el código de Polylla para poder utilizarla.

Por último, en el caso de *clibgen*, se tiene que el código necesario para generar la librería es casi nulo, teniendo soporte parcial con C++ moderno, pero fallando en que la generación de la librería es más complicada, como también su instalación y uso.

Esta investigación fue acompañada de un *proof of concept* [2], que se realizó durante el ramo de introducción al trabajo de titulo, en el cual se utilizó la *C++ MEX API* para generar una pequeña función MEX. Esto resultó en una pequeña función que convertía mallas de triángulos de un formato a otro en MATLAB.

4.2.1. Diseño de la interfaz de MATLAB

Para genera la interfaz se tiene que considerar en primer lugar que la *C++ MEX API* solo permite generar funciones de MATLAB, por lo que para tener Polylla como una clase en el lenguaje, se programó directamente la clase en MATLAB, definiendo solamente los atributos de esta y dejando espacio para que sus métodos sean funciones MEX.

En un principio la idea era que la clase tuviera la menor cantidad de atributos, teniendo un atributo para la triangulación inicial en formato `triangulation`, otro para dejar la malla de polígonos resultante en formato de arreglo de `polyshape` y un atributo para dejar las métricas obtenidas. Por otro lado, la idea era crear una función MEX para el constructor, en la que se convirtiera desde `triangulation` de MATLAB a `Triangulation` de Polylla, una para la generación de la malla en si misma y una función que convirtiera al formato resultante.

Esta idea se había pensado antes de hacer el proof of concept y una vez hecho este, se pudo corroborar que no era posible de realizar, dado que ninguna de las APIs de C o C++ permite trabajar de buena manera con clases que provee MATLAB (principalmente porque no es posible generar arreglos de objetos de estas clases). Por esto se tuvo que cambiar el esquema de la clase.

El esquema final de la clase considera agregar atributos para los puntos, triángulos e información sobre adyacencia de los triángulos, junto con cambiar el formato de salida de la malla de polígonos a un arreglo de celdas, donde cada celda es un polígono representado por arreglos de los índices de los puntos. Por otro lado, el constructor se implementó completamente en MATLAB y solamente se implementó una función MEX, que toma los nuevos atributos mencionados, crea un objeto de `Triangulation` a partir de ellos, luego crea un objeto `Polylla` y por último convierte la malla de polígonos resultante en el arreglo de celdas mencionado. El código usado fue el siguiente:

```
#include <string>
#include <vector>

#include "mex.hpp"
#include "mexAdapter.hpp"

#include <polylla.hpp>
#include <triangulation.hpp>
#include <vertex.hpp>


class MexFunction : public matlab::mex::Function {

public:

    void operator()(matlab::mex::ArgumentList outputs, matlab::mex::
    ArgumentList inputs) {

        checkArguments(inputs);

        matlab::data::Array polylla = inputs[0];

        matlab::data::Array points = matlabPtr->getProperty(polylla, "points")
    ;
        matlab::data::Array connectivity_list = matlabPtr->getProperty(polylla
    , "triangles");

        matlab::data::ArrayDimensions dims = points.getDimensions();
        int n_points = dims[0];

```

```
28
29          std::vector<Vertex> vertices(n_points);
30          for(int i = 0; i < n_points; ++i) {
31              vertices[i].x = static_cast<double>(points[i][0]);
32              vertices[i].y = static_cast<double>(points[i][1]);
33          }
34
35
36          std::vector<int> triangles(3 * connectivity_list.getDimensions()[0]);
37          for(int i = 0; i < connectivity_list.getDimensions()[0]; i++) {
38              for(int j = 0; j < 3; ++j) {
39                  triangles[3 * i + j] = static_cast<int>(connectivity_list[i][j
    ]) - 1;
40              }
41          }
42
43
44          Triangulation tr(vertices, triangles);
45          Polylla mesh(std::move(tr));
46          mesh.construct_polylla();
47          std::vector<Polygon> polygons = mesh.get_polygonal_mesh();
48
49
50          std::size_t n_polygons = polygons.size();
51          matlab::data::CellArray polygons_output = factory.createCellArray({
    n_polygons, 1});
52
53          for(int i = 0; i < n_polygons; ++i) {
54              auto polygon = factory.createArray<int>({polygons[i].vertices.size
    (), 1});
55              for(int j = 0; j < polygons[i].vertices.size(); ++j) {
56                  polygon[j] = (polygons[i].vertices[j]) + 1;
57              }
58              polygons_output[i] = polygon;
59          }
60
61          matlabPtr->setProperty(polylla, "polygons", polygons_output);
62
63          outputs[0] = std::move(polylla);
64      }
65
66  private:
67
68      std::shared_ptr<matlab::engine::MATLABEngine> matlabPtr = getEngine();
69      matlab::data::ArrayFactory factory;
70
71      /**
72       * @brief Checks if the inputs are valid
73       *
74       * @param inputs A list of inputs
75       */
76      void checkArguments(matlab::mex::ArgumentList inputs) {
77
78          if(inputs.size() != 1) {
79              printError("Polylla just works with a single input");
80          }
```

```
81
82          // Check if the input is a Polylla object with isa('Polylla')
83          matlab::data::Array input = inputs[0];
84          std::vector<matlab::data::Array> isa_args{input, factory.createScalar(
    "Polylla")};
85          matlab::data::TypedArray<bool> isa_check = matlabPtr->feval(u"isa",
    isa_args);
86          if(!isa_check[0]) {
87              printError("Polylla wrapper just works with a Polylla object");
88          }
89      }
90
91      /**
92       * @brief Prints an error message to the MATLAB console
93       *
94       * @param error A string with the message to print
95       */
96      void printError(std::string error) {
97          matlabPtr->feval(u"error", 0, std::vector<matlab::data::Array>({
    factory.createScalar(error)}));
98      }
99
100     void displayOnMATLAB(std::string text) {
101         // Pass stream content to MATLAB fprintf function
102         matlabPtr->feval(u"fprintf", 0, std::vector<matlab::data::Array>({
    factory.createScalar(text)}));
103     }
104 };
```

El código mostrado corresponde a la función MEX, que es una clase con el mismo nombre que hereda de `matlab::mex::Function`. Cuando la función se llama desde MATLAB, se invoca el `operator()`, que es el método en donde se realiza lo mencionado anteriormente. Se tiene una función auxiliar `checkArguments` para asegurarse de que la función MEX no se llama con otro tipo de objeto y un par de funciones para debugging dentro de MATLAB. Es importante notar la existencia de la variable `matlabPtr`, que es un *smart pointer* al engine de MATLAB, lo que permite obtener atributos de objetos y evaluar expresiones. Por otro lado, para generar arreglos de MATLAB se necesita usar la clase ArrayFactory de la *Data API* de C++ de MATLAB.

Con respecto a la estructura del proyecto, MATLAB utiliza como convención que las clases definidas en varios archivos deben estar en una carpeta con el nombre de la clase, agregándole un "@" al inicio, por lo que para utilizar la librería, se debe agregar la carpeta `@Polylla` al *path* de MATLAB o moverla al directorio en donde actualmente se está usando MATLAB. Esta carpeta contiene el archivo polylla.m que contiene la definición de la clase de MATLAB y también el binario correspondiente a la función MEX.

Con respecto a la compilación de la función MEX, se puede realizar desde dentro de MATLAB usando la herramienta `mex`, pero también se agregó una integración con CMake, por lo que no es necesario abrir MATLAB para realizar la compilación de la interfaz .

Por último, con respecto al uso de esta librería, una vez que fue agregada al *path* de MATLAB, se puede llamar a Polylla de 4 maneras:

- Triangulación: se pasa un objeto de tipo `triangulation` o `delaunayTriangulation` con la triangulación a usar
- Solo puntos: se pasa una matriz de $n \times 2$, con los puntos. Polylla se encarga de realizar la triangulación usando la función de Delaunay que provee MATLAB
- Puntos y triángulos: nuevamente se pasa una matriz con los puntos, pero además se puede pasar una matriz de $n \times 3$, que tiene los índices de los vértices que conforman cada triángulo de la triangulación.
- Puntos, triángulos y vecinos: se pasan los mismos objetos que el anterior, pero se suma una descripción de la vecindad, pasando una matriz de $n \times 3$, donde cada fila corresponde a un triángulo e indica los índices de los 3 triángulos adyacentes.

Luego de esto, a partir del mismo objeto `Polylla` generado, se puede sacar la malla resultante del atributo `polygons`. Un código de ejemplo se puede ver como muestra la siguiente figura:

```
points = rand(100, 2);
tr = delaunayTriangulation(points);

p = Polylla(tr);
p = p.construct_polylla();
polygons = p.polygons

```

Figura 4.1: Ejemplo de código usando Polylla en MATLAB

4.3. Python

Como se mencionó en 2.2.3, en Python hay una gran cantidad de formas de generar interfaces desde otros lenguajes, teniendo grandes proyectos que usan cada una de estas para generar sus librerías.

Al igual que en el caso de MATLAB, para discriminar entre las opciones disponibles se utilizaron como criterios que la interfaz debía ser sencilla, fácil de utilizar e instalar y también el soporte a C++ moderno.

La opción seleccionada fue pybind11, dado que cumplía con todos los criterios, incluso siendo compatible con CMake y permitiendo también compilar usando herramientas más clásicas para generar paquetes en Python. Por otro lado, opciones como swig complicaban la generación de la librería; cppyy y ctypes la usabilidad del código generado, mientras que la forma oficial de Python para hacer extensiones usando `Python.h` necesitaba una gran cantidad de código para generar la interfaz y no era compatible directamente con C++ moderno.

La última opción que también se tuvo en cuenta para la decisión final fue Boost.Python, dado que es igual de simple que pybind11. Sin embargo, el principal problema que tiene esta

opción es que está pensada para ser usada con la librería de C++ boost, lo que agrega una dificultad extra para ser compilada y utilizada. En cambio, pybind11 fue pensada como una librería *header only* con el mismo estilo de Boost.Python, pero sin el requerimiento de tener boost.

4.3.1. Diseño de la interfaz de Python

Para generar una interfaz usando pybind11 [12], fue necesario definir el nombre de la extensión y dentro de esta definir cada uno de los elementos que se quieren exponer a Python. Esto se realizó mediante varias macros que provee pybind11, permitiendo exponer clases (atributos y métodos), funciones y variables.

En el caso de Polylla se decidió exponer todas las clases creadas a excepción de `HalfEdge`, dado que esta última solamente se usa dentro de `Triangulation` y se puede manejar mediante los métodos que tiene esta última clase.

Al igual que en la interfaz de MATLAB, se priorizó usar el algoritmo, más allá de poder utilizar todas las clases, por lo cual no se expusieron más métodos de los necesario en `Polylla` y solamente el constructor en `Triangulation`. Dado que `Vertex` y `Polygon` eran estructuras simples sin métodos, se expusieron completamente, con todos sus atributos. Con esto, la interfaz se compone de un archivo `main.cpp` que incluye cada una de las clases de Polylla y luego genera la extensión, definiendo las clases mencionadas anteriormente. El código se puede ver a continuación:

```
#include <pybind11/pybind11.h>
#include <pybind11/stl.h>

#include <polylla.hpp>
#include <triangulation.hpp>

namespace py = pybind11;

PYBIND11_MODULE(Polylla, m) {
    m.doc() = "Polylla: Polygonal meshing algorithm based on terminal-edge "
            "regions"; // optional module docstring

    py::class_<Vertex>(m, "Vertex")
    .def(py::init<>())
    .def_readwrite("x", &Vertex::x)
    .def_readwrite("y", &Vertex::y)
    .def_readwrite("is_border", &Vertex::is_border)
    .def_readwrite("incident_halfedge", &Vertex::incident_halfedge);

    py::class_<Polygon>(m, "Polygon")
    .def(py::init<>())
    .def_readwrite("vertices", &Polygon::vertices)
    .def("__str__", [](const Polygon &p) {
        std::stringstream ss;
        std::string sep = "";
        ss << "Polygon: [";
        for(auto v : p.vertices) {
```

```
            ss << sep << v;
            sep = ", ";
        }
        ss << "]";
        return ss.str();
    });

    py::class_<Triangulation>(m, "Triangulation").def(py::init<std::vector<
    Vertex> &, std::vector<int>>());


    py::class_<Polylla>(m, "Polylla")
    .def(py::init<const Triangulation &>())
    .def("construct_polylla", &Polylla::construct_polylla)
    .def("get_polygonal_mesh", &Polylla::get_polygonal_mesh);
}
```

Se puede ver que utilizando pybind11, el código de la interfaz resulta bastante minimalista. Se puede ver que se parte utilizando la macro `PYBIND11_MODULE`, que permite definir la librería como tal bajo el nombre de `Polylla`. Luego de esto se agregan las clases usando `pybind11::class_` y luego agregando atributos o métodos usando `.def` o `.def_readwrite` con los parámetros necesarios. Notar que la definición de métodos también permite el uso de lambdas, por lo que se agrega de manera simple un método `__str__` para la clase `Polygon`, siendo que la clase original no lo tenia.

Con respecto a la compilación, se utilizó la integración con CMake en un primer momento, dado que era la forma más rápida y cómoda de compilar el código, sin embargo, pybind11 también ofrece la *Pythonic way* de crear paquetes, utilizando *setuptools* y *wheel*, por lo que se terminó agregando este tipo de integración, que permite incluso que la librería se instale con `pip install .`

Por último, para utilizar esta interfaz se debe crear un objeto `Triangulation`, que necesita una lista de `Vertex` y una lista de índices que señalan los índices de los vértices. Luego de esto se puede generar un objeto de la clase `Polylla` y llamar al método `construct_polylla`. Por último, para obtener la lista de los polígonos, se tiene que llamar al método `get_polygonal_mesh` y con eso ya se tiene la malla deseada. Se puede ver el código de ejemplo utilizando una triangulación de Delaunay de la librería científica Scipy a partir de puntos generados aleatoriamente:

```
            points = [[random(), random()] for _ in range(10)]
            tri = Delaunay(points)

            polylla_points = [Vertex() for _ in points]
            for i in range(len(points)):
                polylla_points[i].x = points[i][0]
                polylla_points[i].y = points[i][1]
            triangles = tri.simplices.flatten().tolist()

            triangulation = Triangulation(polylla_points, triangles)
            polylla = Polylla(triangulation)
            polylla.construct_polylla()
            polygons = polylla.get_polygonal_mesh()

```

Figura 4.2: Ejemplo de código usando Polylla en Python

Notar que dado que se necesita una lista de `Vertex`, se debe realizar la conversión entre la lista inicial y la que puede aceptar `Triangulation`.

Capítulo 5

Resultados

Para validar la solución que se diseñó e implementó se decidió realizar pruebas para verificar correctitud y desempeño del código en relación a la implementación original de Polylla.

Para verificar correctitud de las implementaciones se generaron varias triangulaciones a partir de puntos aleatorios, usando la herramienta *triangle* [20]. Las mallas generadas se realizaron con cantidades distintas de puntos para asegurarse de cubrir la mayor cantidad de casos. Las versiones comparadas deben obtener una malla resultante igual a la que genere Polylla original.

Para verificar desempeño o rendimiento del código, se utilizaron las mismas triangulaciones de la parte anterior, pero ahora en vez de buscar un mismo output, se realizaron mediciones de tiempo de ejecución. Las implementación de C++ moderno debe tener tiempos similares a la implementación original, mientras que las interfaces deben tener tiempo cercanos a la implementación de C++.

Para generar los puntos aleatorios se utilizó un script de Python, originalmente por Sergio Salinas para generar archivos `.node` con puntos aleatorios. Este script se corrió con distintas cantidades de puntos, generando 100 archivos por cada tamaño, siendo estos los siguientes valores: 10, 10^2, 10^3, 10^4, 10^5 y 10^6 puntos.

Los archivos `.node` fueron entregados a la herramienta *triangle* para generar la triangulación de Delaunay, entregando archivos `.ele` y `.neigh`. Estos archivos se utilizaron y fueron leídos por cada una de las versiones a comparar.

Las comparaciones se realizaron en un notebook con procesador Intel i77700HQ a 2.8GHz, con 16 GB de memoria RAM. El sistema operativo que se usó fue Windows 10, utilizando MSVC 14.32 como compilador de C++, MATLAB R2022A y Python 3.10.

5.1. Comparación con la versión original

Para realizar la comparación entre la versión original y la implementación usando C++ moderno se tomaron ambas versiones y se ejecutaron sobre todos los archivos de las triangulaciones generadas con *triangle*, guardando los archivos en formato OFF de la malla resultando para compararlos. Junto a esto se recolectó el tiempo que demoraron ambas versiones desde que se terminaba de cargar la triangulación hasta que se terminara de generar la malla de polígonos, sin contar el tiempo de carga de la triangulación en sí, por no ser parte del algoritmo.

Para ejecutar la versión original de Polylla sobre cada uno de los archivos se utilizó un pequeño *script* en *PowerShell*, que permitió correr el ejecutable de Polylla varias veces con los distintos archivos, sin tener que hacerlo a mano uno a uno. Esto se debe a que el código no permite ser usado como librería, por lo que para automatizarlo usando C++ se debía modificar el código y esto hubiera sido más complejo de realizar. Con respecto a tomar el tiempo de ejecución, dado que se quería excluir la carga de la triangulación y la escritura de la malla de polígonos, se agregaron al código un par de lineas que permiten esta medición y que muestran el tiempo en la salida estándar, de manera que el *script* antes mencionado pueda recolectarlo.

Por otro lado, para ejecutar Polylla-Mesh-DCEL en todas las triangulaciones que se necesitaba, se creó un pequeño archivo de C++ que utilizaba la librería creada y que podía ejecutar el algoritmo sin problemas sobre todas las triangulaciones, midiendo los tiempos de ejecución y guardándolos en un formato simple para poder ser graficados después.

Con respecto a los resultados obtenidos, es importante mencionar en primer lugar que se observó que la implementación original dejaba de funcionar con una mayor cantidad de puntos, principalmente por errores de precisión que no permitían que se generara correctamente la malla. Esto produjo que unas pocas mallas no se generaran para las triangulaciones con pocos puntos, una gran cantidad para 10^5 puntos y todas con 10^6 puntos. La cantidad de mallas generadas se pueden observar en la tabla 5.1 mostrada a continuación.

Cantidad de puntos	Cantidad de mallas generadas
10	100
10^2	100
10^3	97
10^4	75
10^5	4
10^6	0

Tabla 5.1: Cantidad de mallas generadas por la implementación original según cantidad de puntos de las triangulaciones utilizadas

Luego de esto, al revisar las primeras triangulaciones se observó en varios casos que las mallas resultantes eran las mismas, pero al ser escritas en archivos los polígonos tenían diferencias en el orden de los vértices, siendo rotaciones distintas de los vértices, por lo que la comparación de archivos OFF resultantes tuvo que ser refinada. La nueva comparación entre

archivos se basó principalmente en comparar polígonos, agregando el chequeo de rotaciones entre ellos y comparando la cantidad de polígonos iguales con la cantidad de polígonos totales de cada malla.

Los resultados de esta revisión mostraron que las mallas resultantes difieren en gran cantidad mientras más puntos se tienen en la triangulación. En la tabla 5.2 se puede ver cómo desde los 1000 puntos en adelante ningún par de mallas son iguales. A partir de esto se tomaron algunas mallas con pocos puntos que no fueran iguales y se realizó un análisis de las diferencias entre estas. En la figura 5.1 se puede ver como la diferencia entre una de las mallas de 10 puntos es simplemente una única arista.

Cantidad de puntos	Cantidad de mallas iguales
10	98
10^2	59
10^3	0
10^4	0
10^5	0
10^6	0

Tabla 5.2: Cantidad de mallas resultantes iguales para ambas implementaciones de Polylla

Figura 5.1: Mallas diferentes generadas por la implementación nueva (izquierda) y la versión original (derecha) usando la misma triangulación inicial

La explicación de la diferencia de aristas, que también sucede en todas las otras triangulaciones, tiene que ver principalmente con los *barrier-edge tips*. La arista que cambia entre las dos mallas de la figura 5.1 es parte de la fase de reparación o eliminación de los *barrier-edge tips*. Se revisó el código y efectivamente esto se debía a una diferencia entre la forma en la que se calculaba la posición de la arista para arreglar el *barrier-edge tip*, sin embargo, ambas conservan la idea del algoritmo original, por lo que no es un problema mayor. Por último, se calculó cuanto era el porcentaje de diferencia promedio entre las mallas, utilizando porcentaje de polígonos coincidentes como métrica, que se muestra en la tabla 5.3. Se puede ver que

efectivamente las mallas son casi iguales a diferencia de los polígonos que fueron generados en la fase de reparación de *barrier-edge tips*.

Cantidad de puntos	Promedio de coincidencia entre mallas (%)
10	99
10^2	96.5
10^3	96.2
10^4	96
10^5	96

Tabla 5.3: Promedio de coincidencia porcentual de polígonos entre mallas generadas por ambas implementaciones de Polylla

Por último, en la tabla 5.4 se pueden ver los tiempos promedios de cada grupo de triangulaciones, junto con la desviación estándar. Se puede observar en la figura 5.2 que la nueva implementación se aproxima a los tiempos de la original, pero tiene un costo mayor para operaciones pequeñas.

	Polylla original		Polylla C++ moderno	
Cantidad de puntos	Promedio (ms)	Desviación estándar (ms)	Promedio (ms)	Desviación estándar (ms)
10	0.029	0.006	1.840	0.439
10^2	0.148	0.034	2.805	0.728
10^3	1.237	0.092	4.153	1.004
10^4	12.124	0.543	15.306	5.622
10^5	122.535	1.397	135.877	13.200
10^6	-	-	1618.855	92.553

Tabla 5.4: Tiempos de ejecución promedio de la implementación de C++ moderno y original en función de la cantidad de puntos

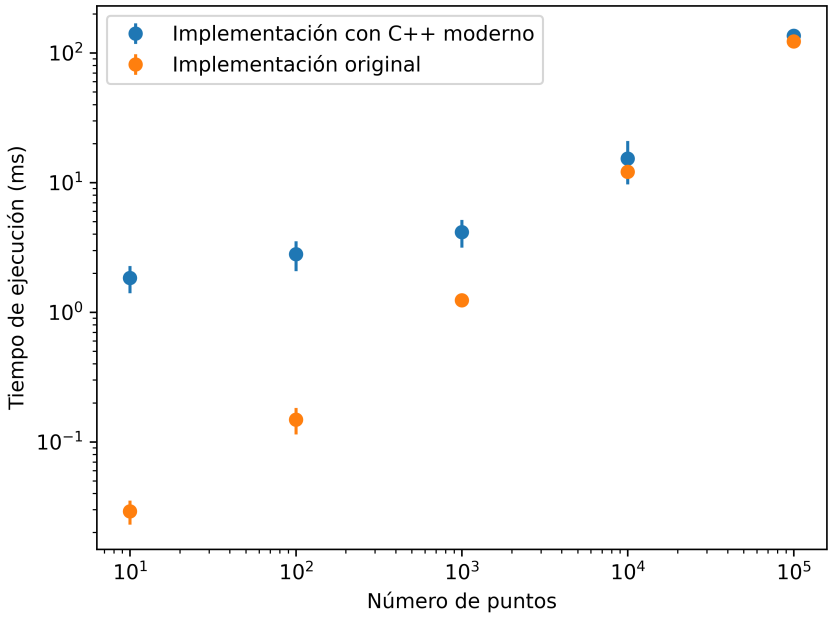

Figura 5.2: Tiempos de ejecución promedio de implementación de C++ moderno y original

5.2. Comparación con interfaz de MATLAB

Al igual que con la comparación entre la versión original y la implementación de C++ moderno, se leyeron los archivos de las triangulaciones previamente generadas y se generaron las mallas de polígonos correspondientes. Se realizó el mismo proceso de chequeo de las mallas resultantes, comparándolas con la mallas que generó la versión de C++ moderno, sin embargo, dado que la interfaz de MATLAB está hecha a partir de la librería que provee la implementación de C++ moderno, entonces la coincidencia fue 100 % sin ningún problema.

Con respecto a los tiempos de ejecución de esta versión, se midieron desde MATLAB, usando `tic` y `toc` midiendo el funcionamiento completo de la función MEX. Los tiempos se muestran en la tabla 5.5 y se puede observar en la figura 5.3 una comparación gráfica con la implementación en C++ sobre la que está hecha la interfaz. Se puede observar que la interfaz de MATLAB funciona a la misma velocidad que la versión de C++ moderno para pocos puntos, pero a medida de que estos aumentan el tiempo empieza a llegar a 10 veces el tiempo de la otra versión, con tendencia a seguir subiendo.

Cantidad de puntos	Promedio (ms)	Desviación estándar (ms)
10	1.801	0.385
10^2	2.599	0.535
10^3	11.428	0.995
10^4	88.537	6.922
10^5	1029.558	156.214
10^6	11374.377	387.863

Tabla 5.5: Tiempos de ejecución promedio de la interfaz de MATLAB en función de la cantidad de puntos

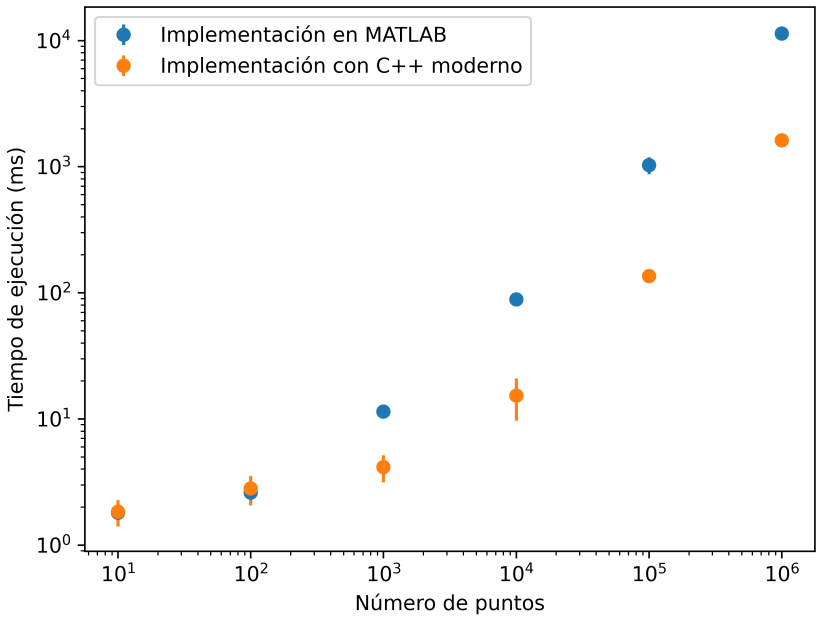

Figura 5.3: Tiempos de ejecución promedio de implementación de C++ moderno e interfaz de MATLAB

5.3. Comparación con interfaz de Python

En el caso de la interfaz de Python, al igual que con las otras versiones, se leyeron los archivos con las triangulaciones y se convirtieron a mallas de polígonos usando la interfaz generada, escribiendo los resultados a archivos OFF. Todo esto fue realizado al igual que con MATLAB, se realizó una comprobación de que las mallas fueran iguales entre esta versión y la de C++, logrando también un 100 % de coincidencia.

En el caso del tiempo de ejecución, se utilizó la función `time.time` de Python, para tomar el tiempo justo antes de llamar a `construct_polylla` y después de este. Los tiempos se muestran en la tabla 5.6 y se pueden ver comparados a la implementación de C++ en la

figura 5.4. Se puede observar que a pesar de que para puntos iniciales se tiene un mayor costo que la versión de C++, a medida que la cantidad de puntos por triangulación aumenta, los tiempos de ejecución de la interfaz mejoran y se acercan a los que tiene la versión de C++.

Cantidad de puntos	Promedio (ms)	Desviación estándar (ms)
10	4.234	4.623
10^2	3.19	2.938
10^3	5.976	2.297
10^4	18.911	1.804
10^5	139.565	18.099
10^6	1779.545	123.566

Tabla 5.6: Tiempos de ejecución promedio de la interfaz de Python en función de la cantidad de puntos de cada triangulación

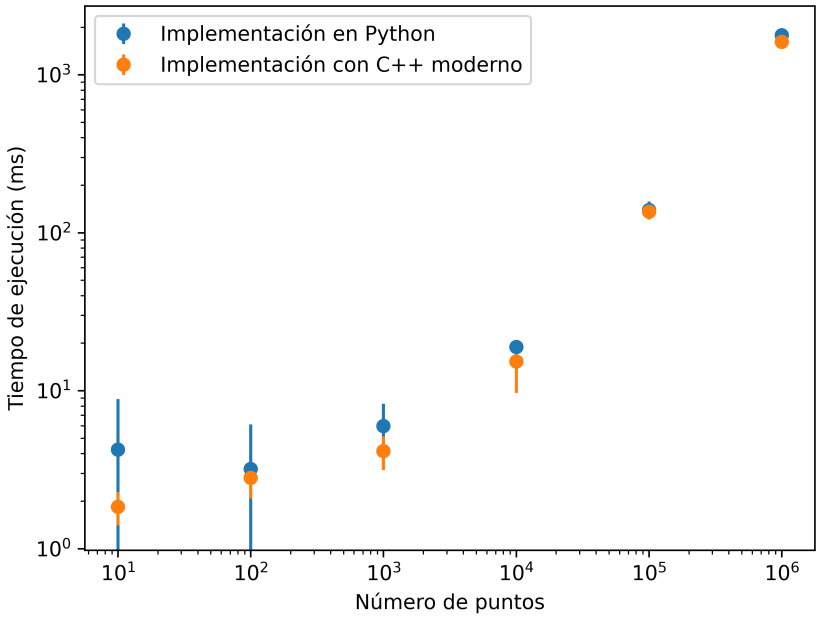

Figura 5.4: Tiempos de ejecución promedio de implementación de C++ moderno e interfaz de Python

5.4. Caso de uso real

Además de las validaciones mencionadas en las secciones anteriores, se realizó una última validación de la interfaz de MATLAB. Esta validación consistió en integrar la interfaz de MATLAB a un código ya existente que usaba la versión antigua de Polylla.

El código mencionado corresponde a parte de la tesis de doble titulación de Sebastián Luza, alumno del departamento de ingeniería civil mecánica. Esta tesis trata sobre mallas

poligonales y el método del elemento virtual, teniendo integrada a la versión original de Polylla en el código.

Para utilizar Polylla y generar mallas de polígonos, la versión final del código de Sebastián necesitaba tener el ejecutable de Polylla compilado externamente en un directorio para su uso. Las triangulaciones que se querían convertir a mallas de polígonos debían ser escritas en archivos `.node`, `.ele` y `.neigh`, luego realizar una llamada al sistema desde MATLAB para ejecutar el binario de Polylla con los nombres de los archivos y por último leer la malla resultante de un archivo. Esto generó que el código tuviera una gran cantidad de código solamente dedicado a la utilización de Polylla, siendo que esta herramienta no se usaba en todo el espacio a mallar, sino que en casos muy específicos.

Para realizar la integración de la interfaz de MATLAB de Polylla a el código ya existente se tomó la carpeta `@Polylla` que contenía el archivo que definía la clase y la función MEX, y se agregó al proyecto. Agregar Polylla al código resultó en un código parecido a las últimas lineas de la figura 4.1, dado que el código escribía en archivos una triangulación de la clase `triangulation`, que es completamente compatible con la interfaz, mientras que el output que genera polylla es suficiente para su uso directo. Esto reemplazó aproximadamente 100 lineas de código que solamente escribían y leían archivos, y también realizaban la llamada al binario de Polylla.

Capítulo 6

Conclusiones

En esta memoria se desarrollaron varias versiones de un algoritmo de conversión de triangulaciones a mallas de polígonos, con el fin de extender una implementación ya existente a varios lenguajes de programación. Se generó una librería de C++ para poder ser utilizada desde otros códigos, un ejecutable que permite usarlo desde línea de comandos, una librería para que se pueda usar desde MATLAB y otra para que se pueda utilizar desde Python.

Las librerías generadas permiten la utilización del algoritmo con mayor facilidad y en un mayor rango de lenguajes de programación, siendo una mejora considerable a la implementación original. Esto cumple el objetivo general propuesto al inicio de este trabajo.

Con respecto a la librería de C++ generada, se tiene que es una gran mejora de la versión anterior, dado que permite ser utilizada como librería (cosa que la versión original no tenía), es extensible y robusta, logrando funcionar con triangulaciones con una cantidad mayor de puntos que la implementación original. Es importante señalar que las mallas no son exactamente iguales a la versión original, ya que difieren levemente en la implementación de la fase de reparación, por lo que las mallas resultantes no van a ser idénticas a las generadas por la versión antigua, sino que difieren en un 4 %. Este porcentaje concuerda con la cantidad de *barrier edge tips* obtenidas en las mallas en el articulo original de Polylla [19], validando los resultados obtenidos.

Con respecto a la diferencia en rendimiento entre la versión de C++ y la versión original, se puede concluir que se logró la velocidad requerida, dado que la versión de C++ moderno se acerca a los tiempos de la original cuando hay una gran cantidad de puntos, sin embargo, existe un costo extra en casos más pequeños. Esto se puede deber a que la versión de C++ moderno tiene que realizar una conversión desde una estructura de datos basada en caras a otra basada en *Half-Edge*, lo que le agrega un costo de tiempo extra y se ve reflejado cuando la cantidad de puntos es menos.

En el caso de las interfaces de MATLAB y Python se generaron librerías compactas y que permiten usar perfectamente el algoritmo sin necesidad de utilizar el binario de Polylla o utilizar archivos para eso. Se puede notar que el rendimiento de la versión de Python es similar al de la versión de C++, mientras que el de MATLAB es menor, pero sigue siendo competente comparándolo con respecto a la versión de C++, obteniendo tiempos de 10 veces

mayores. Esto se puede deber a que se tiene que realizar una conversión desde la estructura de datos de C++ a la estructura de datos de MATLAB, realizando esta operación 2 veces y agregando un costo extra proporcional al número de puntos de la malla.

Además de lo mencionado anteriormente, se logró que todos los códigos y librerías generadas tuvieran mejor documentación que la versión original de la implementación, siendo más sencillas de compilar, instalar y utilizar. Esto permitió que la librería se pudiera integrar a proyectos que ya utilizaban la versión original sin mucha dificultad.

Es importante recalcar que a pesar de que el trabajo realizado logra cumplir los objetivos propuestos al inicio de esta memoria, existe una gran cantidad de trabajo posible a futuro.

En primer lugar, las métricas originales de Polylla no están contempladas en las librerías generadas, por lo que se podrían agregar y con esto tener más formas de comparar las mallas obtenidas.

En segundo lugar, existen más lenguajes de programación en los que seria una buena idea tener Polylla como librería, por su uso en el área de la computación gráfica, uso científico e incluso por su popularidad, por lo que un trabajo futuro podría considerar seguir extendiendo Polylla a más lenguajes como C#, Octave, Julia y Javascript.

Por último, Polylla tiene varias estructuras de datos propias para funcionar, sin embargo, cada librería grande de computación científica o de geometría computacional también tiene las suyas. Esto hace surgir la posibilidad de generalizar la librería de Polylla más aún para permitir su uso con estructuras de datos externas. Un ejemplo de esto podría ser la librería CGAL, Scipy y GAlgebra.

Bibliografía

[1] Accountalive. Dcel halfedge connectivity, 2011. Own work, CC BY-SA 3.0.

[2] Beltrán Amenábar. Matlab mex c++ proof of concept. `https://github.com/beltranamenabar/matlab-mex-cpp-poc`, 2021.

[3] Beltrán Amenábar. Polylla-matlab: A simple wrapper for polylla using c++ mex functions. `https://github.com/beltranamenabar/Polylla-Matlab`, 2022.

[4] Beltrán Amenábar. Polylla-mesh-dcel: Polylla's algorithm as a library. `https://github.com/beltranamenabar/Polylla-Mesh-DCEL`, 2022.

[5] Beltrán Amenábar. Polylla-python: A simple python wrapper for polylla using pybind11. `https://github.com/beltranamenabar/Polylla-Python`, 2022.

[6] The Geometry Center. Geomview. `http://www.geomview.org/`.

[7] L. Beirao da Veiga, F. Brezzi, and L. D. Marini. Virtual elements for linear elasticity problems. *SIAM Journal on Numerical Analysis*, 51(2):794–812, 2013.

[8] Boris Delaunay. Sur la sphère vide. *Bulletin de l'Académie des Sciences de l'URSS, Classe des Sciences Mathématiques et Naturelles*, pages 793–800, 1934.

[9] Nü es. Delaunay voronoi, 2011.

[10] Nü es. Delaunay circumcircles centers, 2012.

[11] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, apr 1985.

[12] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[13] D.E. Muller and F.P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.

[14] Ean Tat Ooi, Chongmin Song, F. Tin-Loi, and Z.J. Yang. Polygon scaled boundary finite element for crack propagation modelling. *International Journal for Numerical Methods in Engineering*, 91:319–342, 07 2012.

[15] María-Cecilia Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *International Journal for Numerical Methods in Engineering*, 40(18):3313–3324, 1997.

[16] Maria-Cecilia Rivara. Lepp-bisection algorithms, applications and mathematical properties. *Applied Numerical Mathematics*, 59(9):2218–2235, 2009.

[17] Sergio Salinas. Polylla-mesh-dcel: A newer and simpler version of polylla using oop and halfedge data structure. `https://github.com/ssalinasfe/Polylla-Mesh-DCEL`, 2021.

[18] Sergio Salinas. Polylla-mesh: Polygonal meshing algorithm based on terminal-edge regions. `https://github.com/ssalinasfe/Polylla-Mesh`, 2021.

[19] Sergio Salinas, Nancy Hitschfeld, Alejandro Ortiz-Bernardin, and Hang Si. POLYLLA: polygonal meshing algorithm based on terminal-edge regions. *CoRR*, abs/2201.11925, 2022.

[20] Jonathan Richard Shewchuk. Triangle: A two-dimensional quality mesh generator and delaunay triangulator. `https://www.cs.cmu.edu/~quake/triangle.html`.

[21] Hang Si. Detri2. `https://www.wias-berlin.de/people/si/detri2.html`.

[22] Chongmin Song. *Automatic Polygon Mesh Generation for Scaled Boundary Finite Element Analysis*, chapter 4, pages 149–207. John Wiley & Sons, Ltd, 2018.

[23] B. Stroustrup. *A Tour of C++*. C++ In-Depth Series. Pearson Education, 2018.

[24] Cameron Talischi, Glaucio H. Paulino, Anderson Pereira, and Ivan F. M. Menezes. PolyMesher: a general-purpose mesh generator for polygonal elements written in matlab. *Structural and Multidisciplinary Optimization*, 45(3):309–328, January 2012.

[25] Thomazthz. Diagrama de voronoi, 2014. Own work, CC BY-SA 4.0.

ANEXOS

Anexo A

Formatos de archivos

En este anexo se detalla información sobre el tipo de archivos usados en Polylla y el formato que tienen. Para más información se puede consultar la página oficial de *triangle* [20] o de Geomview [6].

A.1. Node

Este tipo de archivo especifica los vértices de una malla, identificando cada uno con un índice y su posición en el plano o espacio.

El formato es el siguiente:

- Primera línea: # <Nº vértices><Nº dimensiones><Nº atributos><si utiliza boundary markers o no>
- Siguientes lineas: <índice del vértice><valor en x><valor en y>[<atributo 1><atributo 2>...] [<boundary marker>]

El *boundary marker* sirve para saber si ese vértice es parte del límite del espacio en el que están los vértices. Por otro lado, los atributos corresponden a valores extras, generalmente numéricos, que se pueden guardar en cada uno de los vértices.

A.2. Ele

Este tipo de archivo especifica los triángulos de una malla, identificando cada uno a partir de los índices de los vértices. Se debe utilizar en conjunto con un archivo `.node`

El formato es el siguiente:

- Primer línea: <Nº triángulos>3 <Nº atributos>
- Siguientes lineas: <índice del triángulo><1er vértice><2do vértice><3er vértice>[<atributo 1><atributo 2>...]

Notar que al igual que en el caso de archivos *Node*, los atributos corresponden a datos extras que se pueden guardar en los triángulos.

A.3. Neigh

Este tipo de archivo especifica vecindad entre triángulos de una malla. Se debe utilizar en conjunto con archivos `.node` y `.ele`

El formato es el siguiente:

- Primera línea: <Nº triángulos>3
- Siguientes lineas: <índice del triángulo><1er vecino><2do vecino><3er vecino>

A.4. Off

Este tipo de archivo originalmente fue creado para especificar objetos en 3D, pero sirve de todas maneras para guardar mallas geométricas.

El formato es el siguiente:

- Primera línea: OFF
- Segunda línea: <Nº vértices><Nº caras><Nº aristas>
- Las siguientes <Nº vértices>lineas: <valor x><valor y><valor z>
- las siguientes <Nº caras>lineas: <Nº vértices de cara><1er vértice><2do vértice>*dots* <n-esimo vértice>

En el caso de mallas geométricas en 2D basta con agregar coordenada z = 0 y se obtiene el resultado deseado.