UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
CIUDAD - CHILE

# “ANÁLISIS DE REDISTRIBUCIÓN DE NODOS EN UNA MALLA VOLUMÉTRICA PARA MEJORAR SU CALIDAD”

FABIÁN OSVALDO ARANCIBIA BRITO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: CLAUDIO LOBOS
Profesor Correferente: HUBERT HOFFMAN

Octubre - 2022

## DEDICATORIA

Quiero dedicar esta memoria a mis amigos y a mi familia, especialmente a mi padre y madre, quienes me brindaron su apoyo en diferentes de formas.

# AGRADECIMIENTOS

Me gustaría partir agradeciendo a mi familia, en especial a mi padre, que por medio de su esfuerzo y cariño por mantener a la familia de 4 pequeños parásitos, quise darlo todo también, ellos siempre estuvieron apoyándome, escuchándome, con quienes podía contar mis penurias universitarias, a Tany Riveras, quien aunque estando lejos, podía comunicarme con ella para hacerle saber cuando estaba bien o mal. También agradecer a mis compañeros de universidad,"los simios", con quienes pude compartir y apoyarme en todas las dificultades que la universidad nos impuso, también quiero agradecer a todos mis compañeros de Judo, haciendo también mención especial a mi sensei Gastón Vergara, quienes por medio del entrenamiento, me permitieron mantener los estribos controlados semanalmente, dándome esos tiempos de relajo que tanto se necesitan. También quiero agradecer a mi profesor Guía Claudio Lobos, que me dio facilidades, me aconsejó y me guió en el proceso para poder llevar a cabo este documento final.

Desde el año 2015, cuando ingresé a la universidad, creí que sería similar a exigencias conocidas, pero desde primer año, comprendí que no era nada parecido, reprobando y fallando en diferentes ramos, pero este dolor era compartido con mis compañeros que conocí durante los primeros años. Sinceramente hasta yo me sorprendo de haber llegado hasta aquí, y espero que en los años futuros, las relaciones creadas se mantengan para seguir disfrutando de la vida.

# RESUMEN

**Resumen—** En el contexto de la computación gráfica, la utilización de mallas volumétricas generadas por computadora son usadas para simulaciones de diferentes fenómenos.

En la actualidad no existe una forma de equilibrar la calidad de la malla con la semejanza al objeto a representar.

En esta memoria se propone una modificación al algoritmo Jens_Repair, el cuál busca validar y mejorar la calidad de las mallas volumétricas usando la métrica JENS. La modificación busca arreglar conflictos presentes en el algoritmo que impiden que este logre arreglar las mallas volumétricas satisfactoriamente.

**Palabras Clave—** Calidad, Validez, Jacobiano, Métrica

# ABSTRACT

***Abstract—*** In the context of computer graphics, the use of computer generated volumetric meshes are used for simulations of different phenomena.

Currently there is no way to balance the quality of the mesh with the similarity to the object to be represented.

This document propose a modification to the Jens_Repair algorithm, which seeks to validate and improve the quality of volumetric meshes using the JENS metric. The modification seeks to fix conflicts present in the algorithm that prevent it from being able to fix the volumetric meshes satisfactorily.

***Keywords—*** Quality, Validity, Jacobian, Metric

# GLOSARIO

JS: Scaled Jacobian

JR: Jacobian Ratio

JENS: Elements Normalized Scaled Jacobian

FEM: Finite Elements Method

Hex1.m3d: Hexaedro deformación 1

OHex1.m3d: Output Hexaedro deformación 1

Hex2.m3d: Hexaedro deformación 2

OHex2.m3d: Output Hexaedro deformación 2

8Hex1.m3d: Hexaedro de 8 subdivisiones deformación 1

O8Hex1.m3d: Output Hexaedro de 8 subdivisiones deformación 1

8Hex2.m3d: Hexaedro de 8 subdivisiones deformación 2

O8Hex2.m3d: Output Hexaedro de 8 subdivisiones deformación 2

# ÍNDICE DE CONTENIDOS

**RESUMEN** . . . . . IV

**ABSTRACT** . . . . . IV

**GLOSARIO** . . . . . V

**ÍNDICE DE FIGURAS** . . . . . IX

**ÍNDICE DE TABLAS** . . . . . XII

**INTRODUCCIÓN** . . . . . 1

**CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA** . . . . . 2
1.1 Contexto . . . . . 2
1.2 OBJETIVOS DE LA MEMORIA . . . . . 3
1.2.1 OBJETIVO GENERAL . . . . . 3
1.2.2 OBJETIVOS ESPECÍFICOS . . . . . 3

**CAPÍTULO 2: MARCO CONCEPTUAL** . . . . . 4
2.1 MÉTODO DE ELEMENTOS FINITOS (MEF) . . . . . 4
2.2 FORMACIÓN DE ELEMENTOS GEOMÉTRICOS . . . . . 5
2.2.1 COMPONENTES DE UNA FIGURA GEOMÉTRICA . . . . . 5
2.2.2 ELEMENTOS GEOMÉTRICOS BÁSICOS. . . . . 5
2.2.3 MALLAS GEOMÉTRICAS. . . . . 6
2.3 TÉCNICAS DE MALLADO . . . . . 7
2.3.1 OCTREE MESHING TECHNIQUE . . . . . 8
2.3.2 MANUAL MESHING TECHNIQUE . . . . . 9
2.4 ACEPTACIÓN DE UNA MALLA GEOMÉTRICA . . . . . 10
2.4.1 MALLA INVÁLIDA . . . . . 10
2.4.2 CALIDAD DE UNA MALLA . . . . . 10
2.5 MÉTRICAS DE VALIDACIÓN DE MALLAS Y MEJORAMIENTO DE CALIDAD . . . . 10
2.5.1 SCALED JACOBIAN (JS) . . . . . 11
2.5.2 JACOBIAN RATIO(JR) . . . . . 11
2.5.3 ELEMENT NORMALIZED SCALED JACOBIAN(JENS) . . . . . 11
2.6 VISUALIZADORES DE MALLAS GEOMÉTRICAS . . . . . 12
2.6.1 PARAVIEW . . . . . 13
2.6.2 GEOMVIEW . . . . . 14
2.6.3 Blender . . . . . 15
2.6.4 Maya . . . . . 16
2.6.5 Zbrush . . . . . 17
2.6.6 Cinema 4D . . . . . 18
2.6.7 Curvy3D . . . . . 19

2.6.8 Comparación de softwares 3D . . . 20
2.7 Algoritmos Desarrollados . . . 20
2.7.1 Algoritmo JENS_Repair . . . 20
2.7.2 Algoritmo Viewer . . . 22
2.7.3 Jens_Quality . . . 23
2.8 Sistema operativo . . . 24

**CAPÍTULO 3: PROPUESTA DE SOLUCIÓN** . . . 25
3.1 PROBLEMA A RESOLVER . . . 25
3.2 PROPUESTA DE SOLUCIÓN . . . 27
3.2.1 Separación de nodos . . . 27
3.2.2 Desplazamiento contrario al calculado . . . 28
3.2.3 Implementación de Cascarón . . . 29
3.2.4 Cambio de signo con patrón . . . 31
3.2.5 Desplazamiento inteligente . . . 31
3.2.6 Cambio Proceso iterativo . . . 32
3.2.7 Enfoque de sistema . . . 33
3.2.8 Mover solo uno de los nodos negativos conectados . . . 34
3.2.9 Intercambio de posición de nodos vecinos negativos . . . 34
3.2.10 Desplazamientos de nodos negativos en 2 direcciones . . . 34
3.2.11 Implementación en mejoramiento de calidad . . . 35

**CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN** . . . 37
4.1 Mallas volumétricas usadas . . . 37

**CAPÍTULO 5: CONCLUSIONES** . . . 53
5.1 Conclusiones generales . . . 53
5.1.1 Métricas . . . 53
5.1.2 Desarrollo . . . 53
5.1.3 Visualizadores . . . 53
5.1.4 Perspectiva y análisis . . . 53
5.1.5 Testeo . . . 54
5.1.6 Resultado . . . 54
5.2 Logros alcanzados . . . 55
5.2.1 Objetivos Específicos . . . 55
5.2.2 Objetivo General . . . 56
5.3 Trabajo futuro. . . . 56

**ANEXOS** . . . 58
5.1 Explicación código implementación Jens_Repair . . . 58
5.1.1 estructura . . . 58
5.1.2 Creación de archivo temporal .off para visualización . . . 59
5.1.3 Separar las caras superficiales de la malla volumétrica . . . 60
5.1.4 Obtención de nodos superficiales . . . 61

5.1.5 Cambio proceso iterativo . . . . . 62
5.1.6 Mejorar la calidad del nodo a desplazar y sus vecinos . . . . . 64

**REFERENCIAS BIBLIOGRÁFICAS** . . . . . 68

# ÍNDICE DE FIGURAS


1 Visualización de una simulación FEM de la deformación de un coche tras un choque frontal asimétrico. . . . . 4

2 Elementos Básicos, Jhon Wiley & Sons, Ltd. . . . . 6

3 Malla geométrica de Suzanne separada por la mitad. . . . . 7

4 Malla geométrica de un rodete de turbina. . . . . 7

5 Representación gráfica del método octree. . . . . 8

6 Método octree en malla geométrica. . . . . 8

7 Malla geométrica generada automáticamente vs malla generada manualmente. 9

8 ScreenShot Software ParaView 5.9.1. . . . . 13

9 GeomView software. . . . . 14

10 Screenshot Blender. . . . . 15

11 Screenshot Video maya. . . . . 16

12 Screenshot video Zbrush. . . . . 17

13 Screenshot video Cinema 4D. . . . . 18

14 Screenshot Curvy3D 3.0. . . . . 19

15 Screenshot Terminal ejecutando el algoritmo viewer. . . . . 23

16 Screenshot Terminal ejecutando el algoritmo jens_quality. . . . . 24

17 Screenshot VirtualBox. . . . . 25

18 Hexaedro perfecto con una subdivisión de 8 hexaedros. . . . . 26

19 Hexaedro con subdivisión de 8 con el nodo central desplazado a la superficie superior. . . . . 26

20 Hexaedro de la figura 8 luego de pasar por el programa Jens_repair. . . . . 27

21 malla volumétrica de un cerebro cortado. . . . . 28

22 ejemplo de cambio de dirección de desplazamiento de un nodo. . . . . 29

23 malla superficial de un Hexaedro perfecto. . . . . 30

24 Hexaedro con una división de 8 modificado. . . . . 35

25 Hexaedro de la figura 25 luego de pasar por el programa Jens_repair sin usar proyección. . . . . 36

26 Hexaedro de la figura 12 luego de pasar por el programa Jens_repair usando proyección. . . . . 36

27 Hexaedro de la figura 12 luego de pasar por el programa Jens_repair usando proyección y un threshold alto. . . . . 37

28 Hexaedro modificado 1. . . . . 38

29 Resultado hexaedro modificado 1. . . . . 38

30 Hexaedro modificado 2. . . . . 39

31 Resultado hexaedro modificado 2. . . . . 39

32 Resultado hexaedro modificado 2. . . . . 40

33 Resultado hexaedro modificado 2. . . . . 40

34 Resultado hexaedro modificado 2. . . . . 41

35 Resultado hexaedro modificado 2. . . . . 41

36 eightHexInvModify5 . . . . . 45

37 Output eightHexInvModify5 . . . . . 45

38 Ejecución jens_repair con la malla eightHexInvModify5 . . . . . 45

39 eightHexInvModify6 . . . . . 45

40 Output eightHexInvModify6 . . . . . 45

41 Ejecución jens_repair con la malla eightHexInvModify6 . . . . . 46

42 eightHexInvModify14 . . . . . 46

43 Output eightHexInvModify14 . . . . . 46

44 Ejecución jens_repair con la malla eightHexInvModify14 . . . . . 46
45 eightHexInvModify16 . . . . . 47
46 Output eightHexInvModify16 . . . . . 47
47 Ejecución jens_repair con la malla eightHexInvModify16 . . . . . 47
48 eightHexInvModify21 . . . . . 47
49 Output eightHexInvModify21 . . . . . 47
50 Ejecución jens_repair con la malla eightHexInvModify21 . . . . . 48
51 eightHexInvModify29 . . . . . 48
52 Output eightHexInvModify29 . . . . . 48
53 Ejecución jens_repair con la malla eightHexInvModify29 . . . . . 48
54 eightHexInvModify33 . . . . . 49
55 Output eightHexInvModify33 . . . . . 49
56 Ejecución jens_repair con la malla eightHexInvModify33 . . . . . 49
57 eightHexInvModify40 . . . . . 49
58 Output eightHexInvModify40 . . . . . 49
59 Ejecución jens_repair con la malla eightHexInvModify40 . . . . . 50
60 eightHexInvModify41 . . . . . 50
61 Output eightHexInvModify41 . . . . . 50
62 Ejecución jens_repair con la malla eightHexInvModify41 . . . . . 50
63 eightHexInvModify45 . . . . . 51
64 Output eightHexInvModify45 . . . . . 51
65 Ejecución jens_repair con la malla eightHexInvModify45 . . . . . 51
66 Estructura carpetas algoritmo Jens_Repair. . . . . 58
67 Funciones para generar el archivo temportal .off parte 1. . . . . 59

68 Funciones para generar el archivo temportal .off parte 2. . . . . 59

69 Ejecución de la función para separar las caras superficiales de la malla volumétrica. . . . . 60

70 Función para saber si una cara es superficial. . . . . 60

71 Ejecución de la función projection() en los nodos inválidos superficiales. . . . . 60

72 Función obtener los nodos superficiales parte 1. . . . . 61

73 Función obtener los nodos superficiales parte 2. . . . . 61

74 Función obtener los nodos superficiales parte 3. . . . . 62

75 Función para desplazar los nodos a mejorar parte 1. . . . . 62

76 Función para desplazar los nodos a mejorar parte 2. . . . . 63

77 Función para desplazar los nodos a mejorar parte 3. . . . . 63

78 Función para desplazar los nodos a mejorar parte 4. . . . . 64

79 Función para desplazar los nodos a mejorar parte 5. . . . . 64

80 Función para desplazar los nodos a mejorar parte 1. . . . . 65

81 Función para desplazar los nodos a mejorar parte 2. . . . . 65

82 Función para desplazar los nodos a mejorar parte 3. . . . . 66

83 Función para desplazar los nodos a mejorar parte 4. . . . . 66

84 Función para desplazar los nodos a mejorar parte 5. . . . . 67

85 Función para desplazar los nodos a mejorar parte 6. . . . . 67

# ÍNDICE DE TABLAS

1 Calidades de las mallas utilizadas antes y después de la mejora de calidad. . . . 20

2 Calidades de las mallas utilizadas antes y después de la mejora de calidad. . . . 42

3 Mallas de prueba parte 1. . . . . 43

4 Mallas de prueba parte 2. . . . . 44

# INTRODUCCIÓN

El trabajo con mallas volumétricas generadas por computadora para simulaciones es un área de la informática la cuál puede ser considerada relativamente nueva en comparación con otras áreas de la investigación.

En la actualidad, la generación de mallas volumétricas de forma automática puede generar mallas que presenten problemas, tanto que la malla no sea viable para simulaciones o que esta no se parezca exactamente al objeto que se quiere simular.

Las mallas pueden estar formadas por distintos tipos de elementos, usando los tetraedros, hexaedros, pirámides y prismas.

Para mallas formadas por un subconjunto de los elementos nombrados, existen métricas de mejoramiento de calidad que permiten arreglarlas y que sean viables para simulación. Sin embargo, no existe una forma de corregir las mallas volumétricas de forma estándar para mallas que utilicen cualquiera de los elementos nombrados.

En el presente documento se propone un mejoramiento a un algoritmo utilizado para validar y mejorar la calidad de mallas volumétricas, el cuál utiliza una nueva métrica llamada JENS, con el objetivo de poder trabajar con mallas compuestas por diferentes tipos de elementos geométricos y así obtener resultados que facilitarán el uso de las mallas en simulaciones computacionales de diferentes áreas de la investigación.

La estructura del documento está compuesta por 5 capítulos: En el capítulo 1: Definición del problema, se explica el contexto por el cuál surge la necesidad de innovar en una nueva forma de mejorar las mallas volumétricas y en que sectores se aplicará. En el capítulo 2: Marco conceptual, se presenta la investigación realizada acerca de los contenidos referentes a los temas necesarios utilizados en esta memoria para que el lector pueda ampliar de buena forma su conocimiento y así lograr entender el proceso que se llevó a cabo para la solución planteada. En el capítulo 3: Propuesta de solución, representa todo el desarrollo realizado para lograr llevar a cabo una solución para el problema planteado en este documento, el apartado permite mostrar al lector tanto los avances realizados como los problemas encontrados por el memorista en el proceso. En el capítulo 4: Validación de la solución, se mostrarán las pruebas realizadas, los cuales contienen un análisis visual de las mallas antes y después de ser reparadas y una tabla que contiene los cálculos de las calidades iniciales y finales. Para finalizar, en el capítulo 5: Conclusiones, se exponen las deducciones obtenidas por el memorista tras el desarrollo de la memoria, así como trabajo futuro que se pueda realizar a partir de lo desarrollado..

# CAPÍTULO 1
# DEFINICIÓN DEL PROBLEMA

## 1.1. Contexto

La utilización de mallas geométricas generadas por computadora son una herramienta muy utilizada en áreas como las matemáticas, la mecánica y la biología, siendo una herramienta de no muchos años de desarrollo.

Dentro de la mecánica, éstas se usan para la realización de simulaciones de comportamientos mecánicos, como lo son resistencia de materiales, termodinámica, entre otras.

Una malla puede ser formada por diferentes tipos de elementos, dentro de los cuales se encuentran el hexaedro, prisma, tetraedro y pirámide. En el área de las matemáticas tienden a trabajar con mallas formadas por tetraedros. Sin embargo, para la mecánica o biología, se ha propuesto que las mallas deben estar formadas por los cuatro tipos de elementos para poder obtener una mejor representación del objeto a modelar.

Existen varios algoritmos que permiten la generación de estas mallas de forma automática. Sin embargo, muchas veces las mallas geométricas son generadas con problemas de calidad en los elementos que la conforman, ya que nodos pueden estar posicionados en lugares no óptimos, provocando que caras de la malla se traslapen y generen problemas en su comportamiento en las simulaciones o simplemente, no se pueda simular.

Logrando reposicionar los nodos desde un lugar que provoca conflictos a otro donde sea óptimo se puede lograr que la malla pueda ser usada para simulación. Esto se logra analizando los elementos a los que pertenece el nodo y su posición con respecto a los demás nodos de los elementos analizados.

Para poder validar y mejorar la calidad de estas mallas geométricas existen diferentes tipos de métricas muy utilizadas, como lo son el Js (scaled jacobian) o el JR (jacobian ratio). Estas métricas permiten calcular la dirección de desplazamiento de los nodos que están provocando problemas en la malla. Sin embargo, ninguna de estas métricas puede trabajar con los cuatro tipos de elementos geométricos en su conjunto, esto se debió a que los cálculos de las métricas no funcionan con respecto a ciertos elementos o simplemente no fueron definidos por la complejidad de resolución de estos.

Esto en la actualidad provoca la imposibilidad de las mallas formadas por elementos mixtos de equilibrar la calidad de la malla con la cercanía al modelo representado.

Para poder corregir las mallas mixtas se planteó la métrica JENS (Elements Normalized Scaled Jacobian). Esta métrica basada en el jacobiano escalado busca poder ser aplicada para los cuatro tipos de elementos y así poder desplazar los nodos a corregir.

Terminando el planteamiento del problema y posible forma de solucionar, se hace introducción a los objetivos a cumplir en esta memoria.

## 1.2. OBJETIVOS DE LA MEMORIA

### 1.2.1. OBJETIVO GENERAL

El objetivo de esta memoria es implementar la métrica JENS para redistribuir los nodos de la superficie de una malla geométrica para así mejorar su calidad.

### 1.2.2. OBJETIVOS ESPECÍFICOS

Para poder lograr el objetivo general planteado, se definen los siguientes subobjetivos:

- Generar mallas geométricas con problemas que puedan estar en estado válido o inválido.

  Es necesario poder generar muestras que puedan validar la propuestas de solución.

- Medir la calidad de la superficie de la malla para verificar la validez y calidad de la malla geométrica. Se necesita saber cual es la calidad de la superficie de la malla para saber si se cumple o no la reparación realizada.

- Utilizar la métrica JENS en la superficie de la malla geométrica para desplazar los nodos defectuosos. Como la métrica JENS fue creada para trabajar con los 4 tipos de elementos geométricos, es la que se debe implementar para poder cumplir el objetivo de desplazar los nodos defectuosos.

- Realizar el desplazamiento de nodos y vecinos de forma simultanea para secciones que no puedan ser resueltas moviendo solamente un nodo.

  Como mover un solo nodo puede provocar invalidación en los otros, hay que buscar la forma de que los nodos se muevan de forma simultanea para que esto no ocurra.

- Medir la calidad de la malla geométrica final y verificar su validez. Equivalente al segundo objetivo, permitirá validar si los resultados obtenidos son factibles.

# CAPÍTULO 2
# MARCO CONCEPTUAL

## 2.1. MÉTODO DE ELEMENTOS FINITOS (MEF)

El método de elementos finitos es un método numérico que permite encontrar una solución aproximada para un conjunto de ecuaciones diferenciales parciales que representan los fenómenos a analizar. Publicado en su más reciente versión por Zienkiewicz et al. [Zienkiewicz *et al.*, 2013], el método fue desarrollado originalmente en 1967, hasta llegar a su séptima versión en 2013.

Este método se enfoca en el análisis mecánico estructural y nos podemos referir a éste como un procedimiento general de discretización de problemas continuos planteados por enunciados definidos matemáticamente.

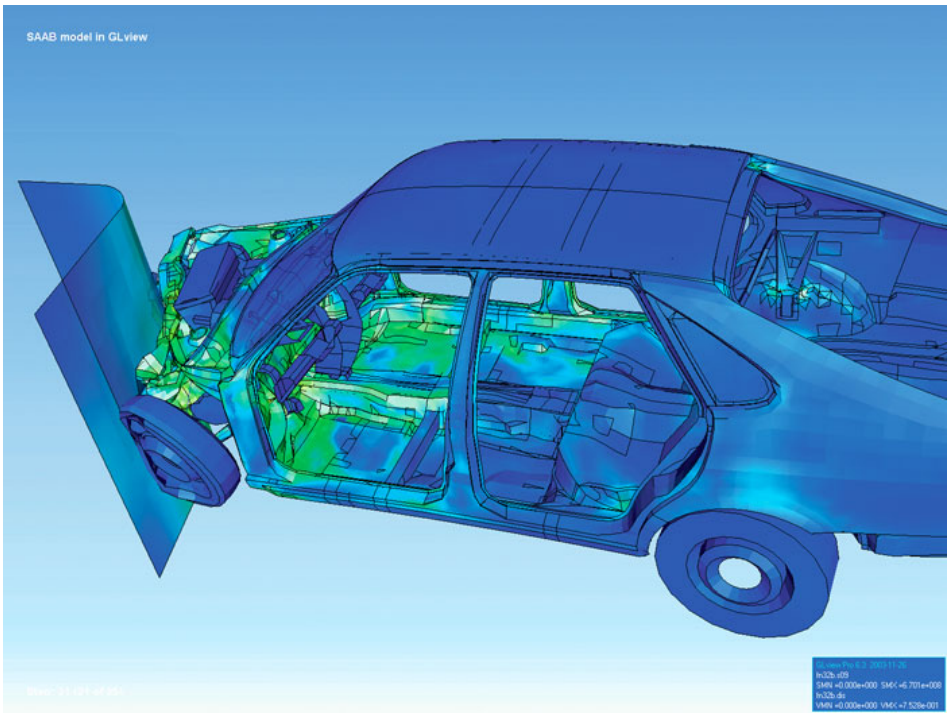


Figura 1: Visualización de una simulación FEM de la deformación de un coche tras un choque frontal asimétrico.

Autor: Maksim.

## 2.2. FORMACIÓN DE ELEMENTOS GEOMÉTRICOS

Para comprender a cabalidad el trabajo desarrollado en esta memoria, es necesario establecer algunos conceptos de geometría básica, para entender la formación de mallas volumétricas, los que se detallan a continuación.

### 2.2.1. COMPONENTES DE UNA FIGURA GEOMÉTRICA

Para formar un elemento geométrico, debemos entender tres conceptos pertenecientes a su estructura:

- Nodo: También conocidos como vértice, es un punto en el cual se intersectan 2 o más líneas rectas.
- Arco: También conocido como puente, es la línea resultante del cruce entre dos superficies o planos. Los arcos también son los segmentos de una recta que marcan el límite de los lados de una figura plana.
- Cara: También conocido como plano, es un objeto bidimensional que es formado por la unión de al menos tres puentes unidos entre sus nodos extremos, formando una figura cerrada.

### 2.2.2. ELEMENTOS GEOMÉTRICOS BÁSICOS.

La presente memoria se desarrolla en el marco de una técnica de mallado particular, definida en [Lobos y González, 2015]. En ésta, se utilizan 4 tipos de elementos, los que se detallan a continuación:

- Hexaedro: Poliedro de 6 caras. Ver Figura 2a.
- Prisma: Poliedro formado por dos triángulos iguales y paralelos en la parte superior e inferior y tres cuadriláteros laterales que los unen. Ver Figura 2b.
- Pirámide: Poliedro formado por una base cuadrada y cuatro triángulos que se interceptan en sus nodos superiores, conocida también como pirámide cuadrangular. Ver Figura 2c.
- Tetraedro: Poliedro formado por una base triangular y tres triángulos que se interceptan en sus nodos superiores, conocida también como pirámide triangular. Ver Figura 2d.

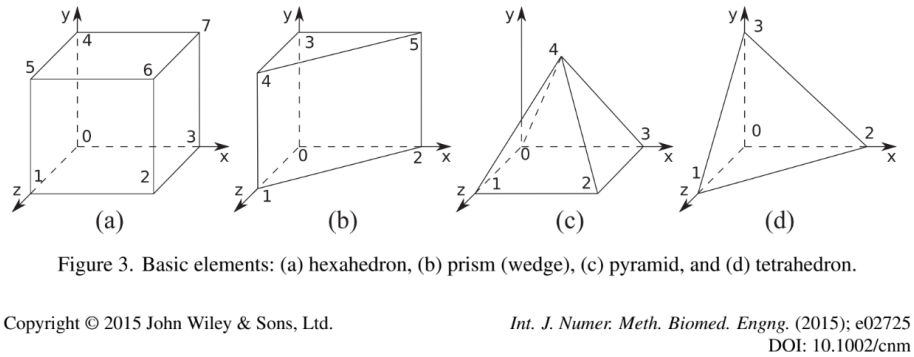


Figura 2: Elementos Básicos, Jhon Wiley & Sons, Ltd.
Fuente: [Lobos y González, 2015].

### 2.2.3. MALLAS GEOMÉTRICAS.

Para una simulación de un objeto mediante computadora, se requiere una función que permita describirlo, la cual muchas veces no existe por la complejidad de la geometría de éste. Es por esto que se debe aproximar a ésta usando los elementos geométricos básicos nombrados anteriormente, esta aproximación recibe el nombre de malla[1].

Existen dos usos principales para una malla:

- Visualización

  Para este uso, los nodos internos del objeto modelado no son de utilidad y por ende solo se usa la superficie de la malla.

  Las mallas superficiales generalmente están formadas por triángulos o paralelogramos, aunque es posible que sean formados por figuras con más arcos, como pentágonos o hexágonos, sin embargo, estos pueden provocar problemas de rendimiento. Estas mallas suelen ser utilizadas en el área de entretenimiento como la animación o videojuegos, ya que en estas áreas no es requerido el uso de comportamientos físicos en el interior de las mallas.

[1]https://www.inf.utfsm.cl/ clobos/meshintroEs.html

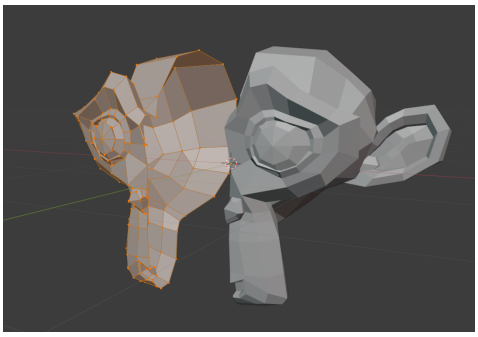

Figura 3: Malla geométrica de Suzanne separada por la mitad.
Fuente: Screen shot blender modelo de Suzanne.

- Simulación

  Para este uso, los nodos internos si son necesarios, llamándose así mallas volumétricas, y estas son utilizadas para la simulación de ecuaciones físicas, para análisis estructurales, deformaciones, fracturas, efectos del calor, entre otras.

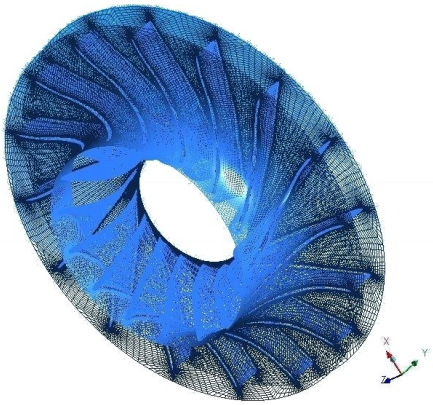

Figura 4: Malla geométrica de un rodete de turbina.
Fuente: Santiago Lain.

## 2.3. TÉCNICAS DE MALLADO

Existen diferentes formas de obtener una malla geométrica, y dependiendo del propósito de ésta, se busca usar una técnica de mallado adecuado que busque generar una malla que cumpla con las características necesarias.

### 2.3.1. OCTREE MESHING TECHNIQUE

La técnica de mallado octree implementa una estructura recursiva compuesta de octantes [Lobos y González, 2015].

Un octante es una región del espacio que tiene intersección de volumen nulo con cualquier otro octante, a excepción de sus hijos.

Generalmente un octante se puede representar como un cubo, pero conceptualmente es cualquier geometría. La única restricción es que un octante debe tener la operación división, esto es, que un elemento pueda ser dividido en ocho elementos iguales.

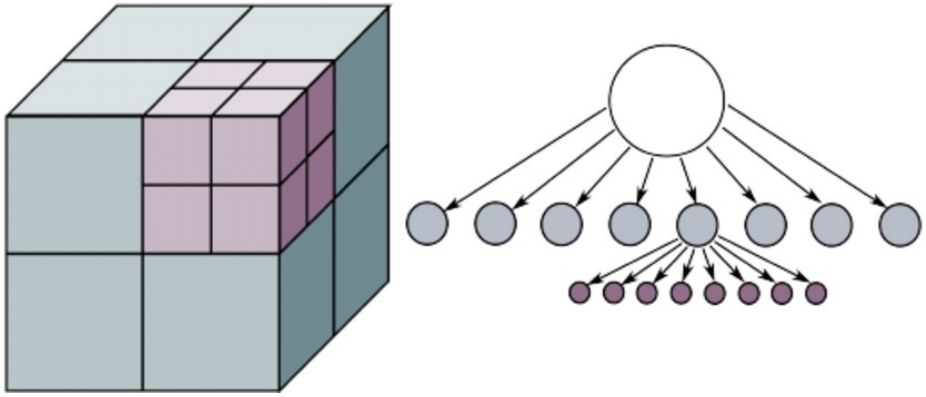

Figura 5: Representación gráfica del método octree.
Fuente: Yuri Vassilevski.

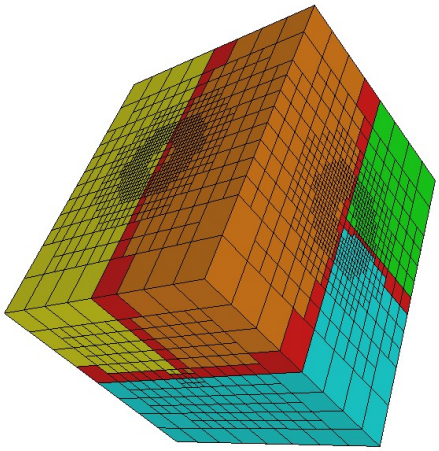

Figura 6: Método octree en malla geométrica.
Fuente: Altair Forum User.

### 2.3.2. MANUAL MESHING TECHNIQUE

Dentro de la biomecánica es muy común la propuesta de mallas FE (finite elements) hechas a mano [Bucki *et al.*, 2010]. Esto se debe a que es importante identificar subregiones asociadas a subestructuras anatómicas en la malla.

Además, se tiende a preferir los hexaedros por sobre los tetraedros, basándose en consideraciones numéricas y por el hecho de la existencia de materiales incompresibles y/o casi incompresibles, los tetraedros con funciones de forma lineal tienden a bloquearse y volverse demasiado rígidos.

Sin embargo, estas técnicas, a pesar de tener mejoras a lo largo del tiempo, requiere una cantidad excesiva de esfuerzo manual.

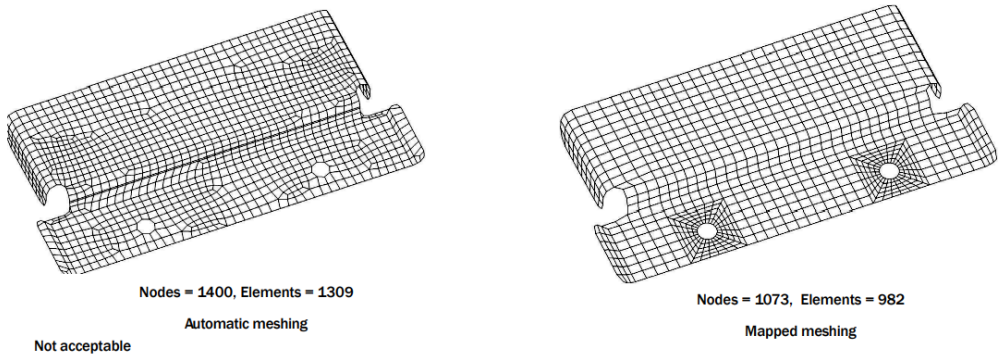


Figura 7: Malla geométrica generada automáticamente vs malla generada manualmente. Fuente: AltairUniversity.

## 2.4. ACEPTACIÓN DE UNA MALLA GEOMÉTRICA

Para que una malla pueda ser utilizada en las simulaciones, primero debemos saber si ésta es válida (que no tiene defectos que generen problemas) y que tenga una calidad aceptable para que la simulación sea lo más representativa posible.

### 2.4.1. MALLA INVÁLIDA

Para saber si una malla es inválida, se chequean todos los elementos que conforman a ésta antes de realizar el análisis FE (finite elements). Si el valor del jacobiano de alguno de los elementos resulta ser negativo, entonces la malla se considera inválida y por tanto no es aplicable el análisis FE [Bucki *et al.*, 2011].

### 2.4.2. CALIDAD DE UNA MALLA

El criterio para medir la calidad de un elemento es utilizando el Jacobian Ratio (JR), cuando un nodo “a” se encuentra muy cerca de un plano formado por sus nodos vecinos, es decir, con quienes forma un arco dentro del mismo elemento, el jacobiano de “a” tendrá un valor cercano a 0, significando así que el elemento tiene una calidad pobre, cuando este valor aumenta, significa que la calidad del elemento también aumenta, siendo 1 el valor ideal. Adoptando el análisis comercial FE, se considerará que un elemento tiene una calidad pobre si JR es menor a 0.03 [Bucki *et al.*, 2011].

La medición de la calidad de la malla tiene 2 propósitos:

- Comparar la malla, ya sea con otras mallas producidas para el mismo dominio o con la propia noción de malla “teóricamente perfecta”para saber que malla se adecua mejor para la simulación.
- Mejorar la malla, es decir, adquirir una malla “mejor” a partir de un estado de mala calidad mediante el uso de algoritmos de reparación de malla.

## 2.5. MÉTRICAS DE VALIDACIÓN DE MALLAS Y MEJORAMIENTO DE CALIDAD

Cuando la malla generada presenta problemas de validez o tiene calidad pobre, es de interés poder utilizar esta malla y validarla o mejorar su calidad, ya que generar mallas hasta encontrar una que cumpla con todas las restricciones del problema puede tomar demasiado tiempo o producir una malla de alto costo computacional en una simulación. Es por esto por

lo que han surgido métricas de calidad que permiten tomar una malla geométrica y corregir los defectos de ésta.

### 2.5.1. SCALED JACOBIAN (JS)

Métrica de calidad basada en el jacobiano. Esta métrica ha sido muy utilizada por varios autores para el mejoramiento de mallas geométricas formadas por hexaedros [C. *et al.*, 2022].

el Js del nodo de índice "i", que llamaremos $J_S^i$ es el jacobiano del tetraedro virtual formado por el nodo "i" y los tres nodos conectados a el en el elemento. Este "jacobiano" debe ser computado usando vectores normalizados empezando desde el nodo "i". Por ejemplo, considerando el hexaedro de la figura 2 con índices 0,....,7 y diremos que las coordenadas de cada nodo están denotados por n0,....,n7 respectivamente. Si se computa el jacobiano escalado para el nodo "i", i.e. $J_S^i$, se define $v_{i,j} = n_j - n_i$ y $\hat{v}_{j,i} = v_{i,j}/||v_{i,j}||$, donde "j" iteraciones sobre los nodos conectados a "i". Con esta formulación podemos computar por ejemplo, el valor para el nodo 0:

$$J_S^0 = \hat{v}_{4,0} \cdot (\hat{v}_{1,0} \times \hat{v}_{3,0}) = \hat{v}_{1,0} \cdot (\hat{v}_{3,0} \times \hat{v}_{4,0}) = \hat{v}_{3,0} \cdot (\hat{v}_{4,0} \times \hat{v}_{1,0}) = 1$$

Con esta formulación, Js $\in [-1, 1]$, dando valor 1 cuando el nodo se encuentra en la posición óptima y si el valor es negativo, el elemento es inválido y no puede usarse en una simulación FEM. Además, si el jacobiano escalado del nodo "i" es menor a 0.2, la calidad del hexaedro es cuestionable.

### 2.5.2. JACOBIAN RATIO(JR)

Métrica de calidad basada en el jacobiano. Esta busca medir el jacobiano para cada nodo "n" de un elemento, tomando el valor máximo de los jacobianos [Kwok y Chen, 2000].

Para cada nodo "n" se calcula la división entre su jacobiano y el jacobiano máximo, usando R como la medida de relación de aspecto. Los valores resultantes obtenido varían entre [0,1], siendo 0 la calidad más pobre hasta 1 como la mejor calidad.

Esta métrica funciona para mallas formadas por hexaedros, prismas y/o pirámides.

### 2.5.3. ELEMENT NORMALIZED SCALED JACOBIAN(JENS)

Es una nueva métrica propuesta por "Claudio Lobos y E.González" basada en scaled jacobian, que busca extender los mismos principios para que pueda ser utilizado no solo por hexaedros, sino también, por tetraedros, pirámides y prismas. [C. *et al.*, 2022].

Se define el Jens para un nodo con índice “i”, i.e. $J^i_{ENS}$ de la siguiente forma:

$$J^i_{ENS} = \begin{cases} (1+k^e) - J^i_S & \text{si } J^i_S > k^e, \\ J^i_S / K^e & \text{si } -k^e \leq J^i_S \leq k^e, \\ -(1+k^e) - J^i_S & \text{si } J^i_S < k^e \end{cases}$$

Donde $K^e$ es una constante definida por el tipo de elemento, excepto para los nodos en caso de pirámides. La constante se obtiene midiendo el $J_S$ de cada nodo de un elemento perfecto.

Para el caso de las pirámides, no se puede aplicar directamente $J_S$ ya que la pirámide presenta un nodo con cuatro conexiones. Por lo que se procede a subdividir en cuatro tetraedros.

## 2.6. VISUALIZADORES DE MALLAS GEOMÉTRICAS

Si bien es posible trabajar con mallas geométricas directamente a través de código y archivos que contienen la información de la estructura de la malla, muchas veces es útil tener la posibilidad de observar visualmente la malla con la que se está trabajando para poder analizar el mejoramiento de esta de una forma más rápida. Es por esto que existen varios programas que permiten la visualización y trabajo de una malla a través de una interfaz.

### 2.6.1. PARAVIEW

ParaView[2] es una aplicación de visualización y análisis de datos multiplataforma de código abierto. Este programa es el más recomendable si se quiere trabajar en Windows.

Con este software se pueden crear visualizaciones rápidamente para analizar sus datos utilizando técnicas cualitativas y cuantitativas.

La exploración de datos se puede realizar de forma interactiva en 3D o mediante programación utilizando las capacidades de procesamiento por lotes de ParaView.

El proyecto ParaView comenzó en el año 2000 en un esfuerzo colaborativo entre Kitware Inc. Y Los Alamos National Laboratory.

La primera versión de ParaView 0.6 fue lanzada en el año 2002 y su última versión ParaView 3.0 fue lanzada en el año 2007.

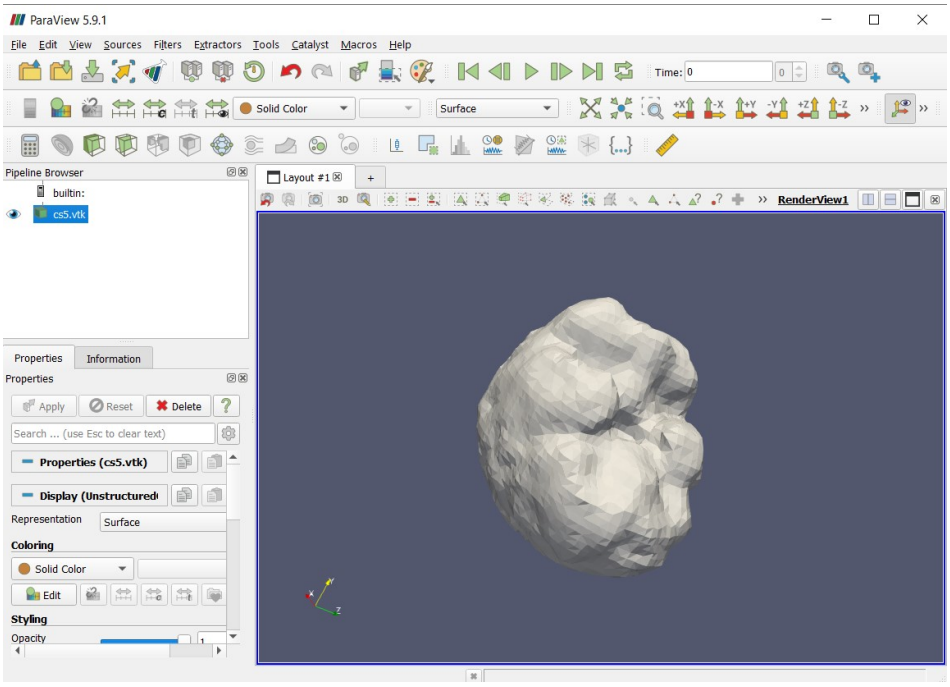


Figura 8: ScreenShot Software ParaView 5.9.1.
Fuente: Elavoración propia.

[2]https://www.paraview.org/

### 2.6.2. GEOMVIEW

Geomview[3] es un programa gratuito de visualización 3D interactivo que permite ver y manipular objetos tridimensionales. también se puede utilizar como un visor independiente para objetos estáticos o como un motor de visualización para otros programas que producen geometría que cambia dinámicamente .

El software es compatible con la mayoría de los sistemas Unix, incluyendo GNU/Linux y Mac OS X. También hay diferentes formas de correrlo bajo Microsoft Windows.

Este software acepta diferentes formatos de archivos y también tiene la capacidad de crear objetos.

En el año 2014 fue liberada su última versión 1.9.5.

Figura 9: GeomView software.
Fuente: Elaboración propia.

[3]http://www.geomview.org/

### 2.6.3. Blender

Blender[4] es una suite de creación 3D enfocada en visualización que ofrece una infinidad de herramientas como lo son modelado, montaje, animación, simulación, renderizado, composición y seguimiento de movimiento.

La primera versión de blender fue lanzada el año 1994. Actualmente cuenta con la última versión 2.93 lanzada el año 2021.

La fundación blender fue fundada en el 2002 por Ton Roosendaal.

Este software es gratuito y de código abierto, por lo que es muy utilizado por independientes y pequeños estudios, siendo así también ideal para estudiantes que busquen entrar en el área de la visualización 3D.

Este software además acepta varios tipos de extensiones de archivos.

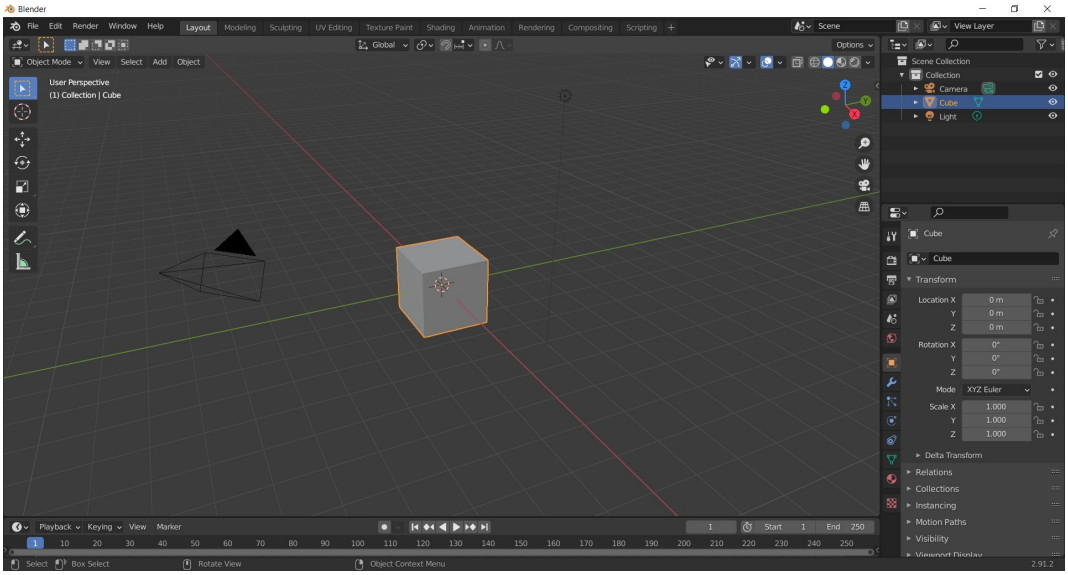


Figura 10: Screenshot Blender.
Fuente: Elaboración propia.

[4]https://www.blender.org/

### 2.6.4. Maya

Maya[5] es un software 3D profesional para crear personajes realistas y efectos dignos de películas taquilleras pertenciente a Autodesk.

La primera versión de Maya fue lanzada el año 2008. Actualmente cuenta con la última versión 2.93 lanzada el año 2022.

Este software es pagado y puede recibir varios tipos de extensiones de archivos

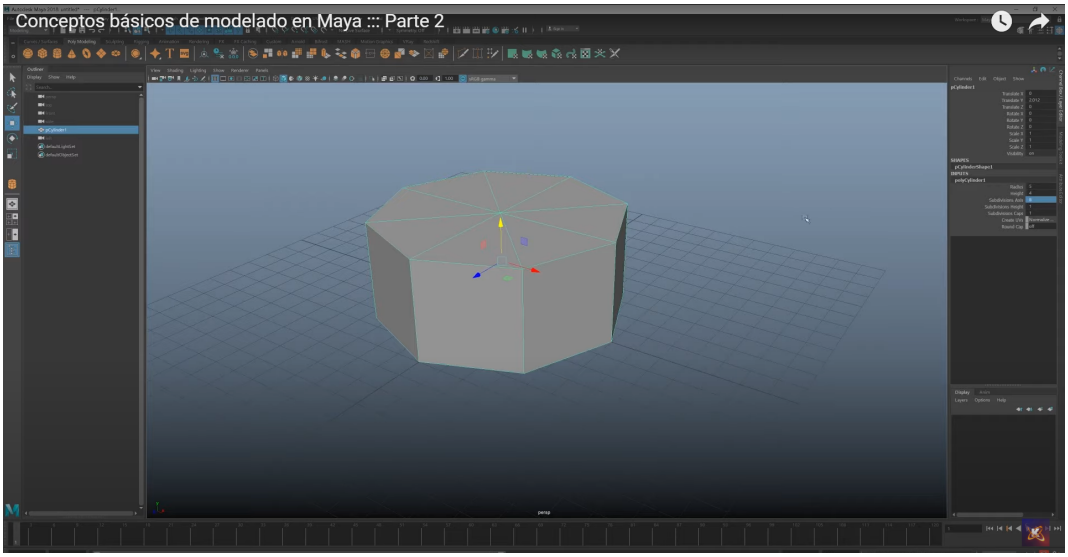


Figura 11: Screenshot Video maya.
Fuente: Elaboración propia.

[5]https://latinoamerica.autodesk.com/products/maya

### 2.6.5. Zbrush

ZBrush[6] establece el estándar de la industria para esculpir y pintar digitalmente. Sus funciones le permiten usar pinceles personalizables para dar forma, texturizar y pintar arcilla virtual en un entorno en tiempo real que brinda retroalimentación instantánea.

ZBrush es un software de pago desarrollado por Maxon, soportado para sistemas MacOs y Windows, pero soporta una menor cantidad de extensiones de archivos.

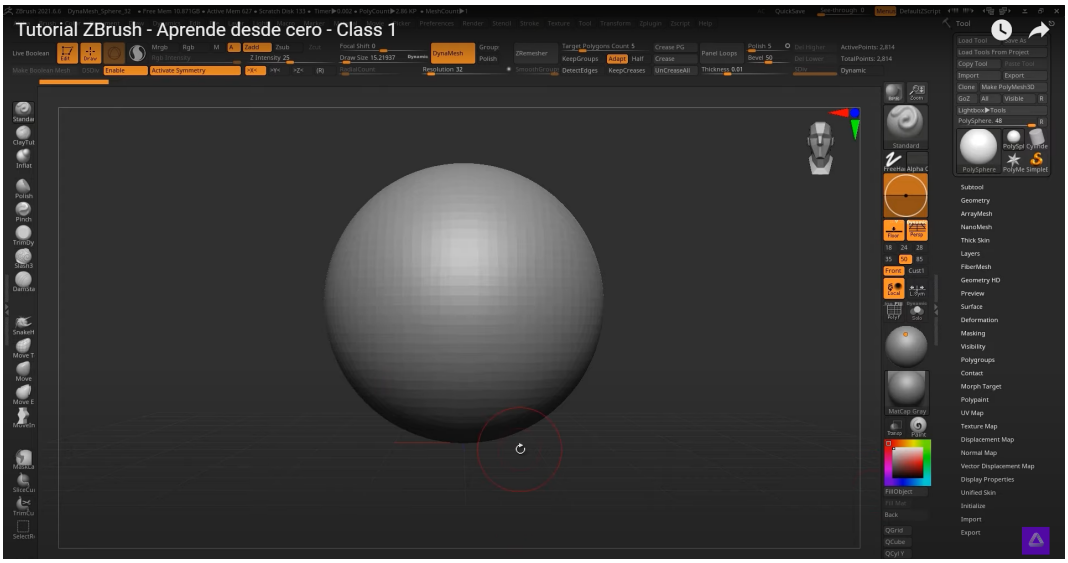


Figura 12: Screenshot video Zbrush.
Fuente: Elaboración propia.

[6]https://pixologic.com/

### 2.6.6. Cinema 4D

Cinema 4D[7] es una solución de software profesional de modelado, animación, simulación y renderizado 3D. Fue desarrollado por Maxon, Su conjunto de herramientas rápido, potente, flexible y estable hace que los flujos de trabajo 3D sean más accesibles y eficientes para los profesionales del diseño, gráficos en movimiento, VFX, AR/MR/VR, desarrollo de juegos y todo tipo de visualización. Cinema 4D produce resultados asombrosos, ya sea trabajando solo o en equipo.

Puede recibir una gran cantidad de extensiones de archivos.

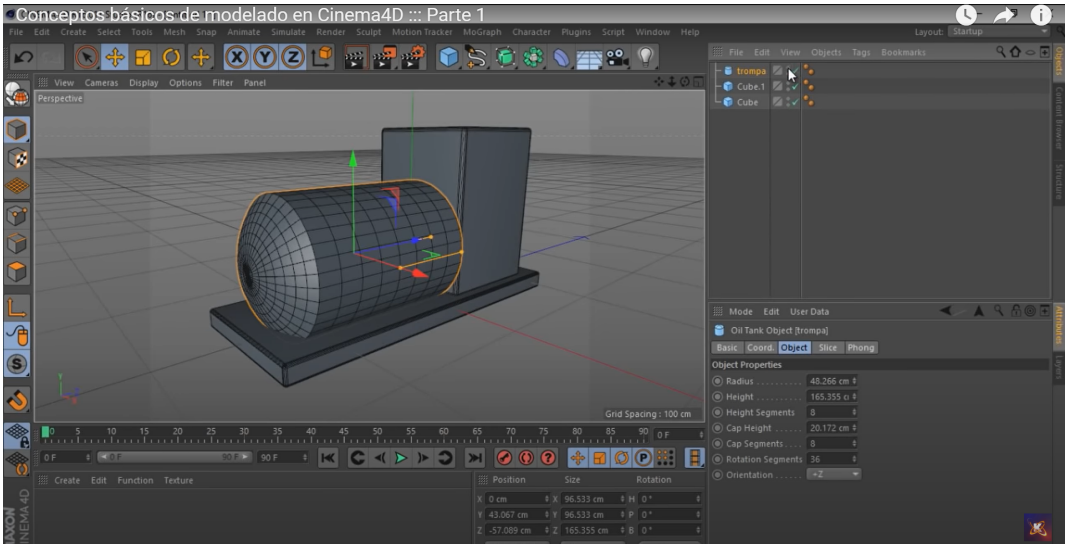


Figura 13: Screenshot video Cinema 4D.
Fuente: Elaboración propia.

[7]https://www.maxon.net/es/cinema-4d

### 2.6.7. Curvy3D

Curvy3D[8] es un software de pago desarrollado por Aartform de esculpido 3D que mediande el dibujo de curvas permite un trabajo suave y rápido para modelos 3D.

La primera versión de Curvy 3D fue lanzada el año 1994. Actualmente última versión de curvy 3D es la 4.0.

Curvy 3D solo corre en el sistema operativo Windows y solo puede importar archivos en formato .obj

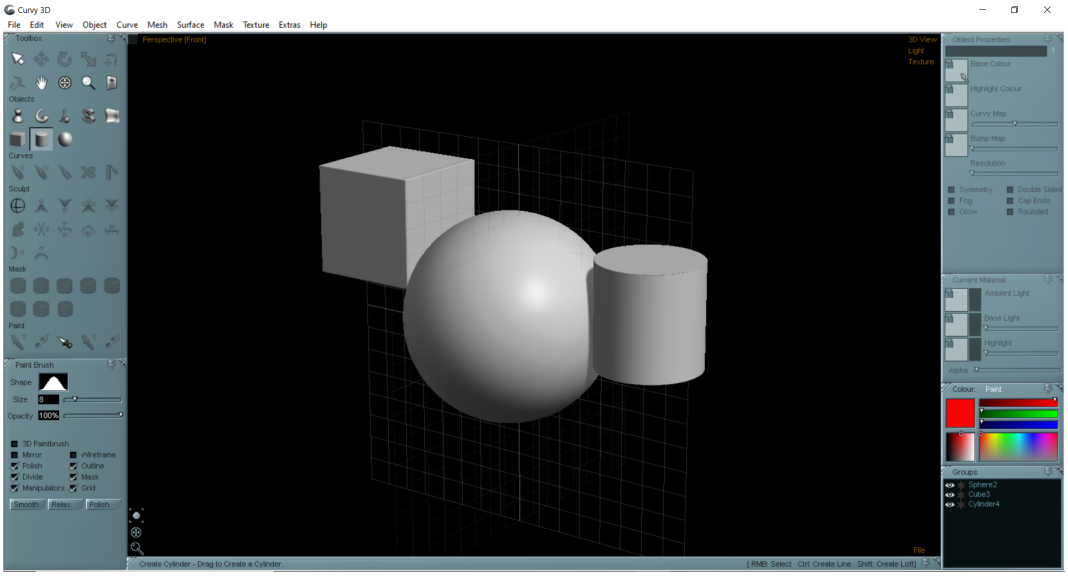


Figura 14: Screenshot Curvy3D 3.0.
Fuente: Elaboración propia.

[8]https://www.curvy3d.com/

### 2.6.8. Comparación de softwares 3D

En esta sección se presenta una tabla comparativa con los programas para trabajar con contenido 3D listados anteriormente:

Tabla 1: Calidades de las mallas utilizadas antes y después de la mejora de calidad. Fuente: Elaboración Propia.

| Software 3D | mallas volumé-tricas | mallas superfi-ciales | Soporte Linux | Soporte Win-dows | Soporte macOs | Pagado |
|---|---|---|---|---|---|---|
| ParaView | Si | Si | Si | Si | Si | No |
| GeomView | Si | Si | Si | No | Si | No |
| Blender | Si | Si | Si | Si | Si | No |
| Maya | Si | Si | Si | Si | Si | Si |
| Zbrush | Si | Si | No | Si | Si | Si |
| Cinema 4D | Si | Si | Si | Si | Si | Si |
| Curvy 3D | No | Si | No | Si | No | Si |

Si bien era posible utilizar distintos programas 3D para la visualización requerida, se optó por utilizar el programa GeomView, esto ya que existe un programa desarrollado llamado Viewer detallado en la siguiente sección que permite cambiar el formato de los archivos de los modelos 3D de forma automática para ser visualizados .

## 2.7. Algoritmos Desarrollados

A continuación se presentarán los algoritmos a utilizar junto con una descripción de la estructura y función de estos. Esto con el objetivo de dar a conocer al lector los algoritmos para entender el como se trabajará en estos.

### 2.7.1. Algoritmo JENS_Repair

Programa desarrollado por Claudio Lobos.

Si bien como se mencionó con anterioridad que existen diferentes métricas para poder mejorar la calidad de las mallas. El objetivo principal es poder mejorar mallas que estén compuestas por los diferentes tipos de elementos nombrados.

Es por esto que como solución se utilizará la métrica jens propuesta para cumplir el objetivo.

El input del programa tiene la siguiente estructura:

./jens_repair input inputSF metric max_iter threshold output

- input: Es una malla volumétrica en formato m3d
- metric: Es la función a usar para la optimización de calidad. Esta puede ser “e” para JENS o “r” para Jacobian Ratio
- max_iter: El número de iteraciones que el algoritmo hará.
- threshold: El valor de calidad que se quiere alcanzar (esta debe ser $\leq 1$).
- output: El nombre del archivo de la malla resultante (la malla resultante está en formato m3d).
- inputSF: Es una malla superficial en formato mdl.

Antes de comenzar estos procesos, primero se debe leer la malla.

En esta sección del algoritmo, la malla es recorrida en cada uno de sus elementos, y estos a su vez, recorridos en cada uno de sus nodos. Para cada nodo se calcula el jacobiano del nodo con respecto a sus nodos vecinos, los cuales pueden pertenecer a diferentes elementos.

Acá se guardarán en una lista todos los nodos que obtuvieron una calidad menor a la ingresada por el usuario (threshold).

Este algoritmo tiene dos partes importantes que son esenciales para poder corregir la malla geométrica que reciba.

- Validación de la malla.

  Para la validación de la malla, vamos a centrarnos en todos los nodos guardados que tengan una calidad menor a 0, esto significa que el/los elementos al que pertenezca este nodo es/son inválidos.

- Mejoramiento de la malla.

  Para el mejoramiento de la malla, vamos a tomar todos los nodos que, después de validar la malla, tengan calidades menores a la ingresada por el usuario.

Para poder llevar a cabo las dos etápas nombradas anteriormente, se requiere mover los nodos de la malla volumétrica.

- Método de desplazamiento de nodos.

  Para desplazar un nodo, se calcula el jacobiano después de desplazar el nodo en $dx$, $dy$ y $dz$, de esta forma, se calcula la dirección de desplazamiento del nodo.

De forma arbitraria, se elige cuanto se desplazará un nodo.

Cada vez que el nodo es desplazado, se calcula el jacobiano para verificar que logra igualar o superar la calidad objetivo (0 para validar, ingresada por el usuario para mejorar), en caso de seguir por debajo, seguirá desplazando el nodo hasta que lo logre.

El algoritmo puede o no entregar una malla de mejor calidad.

### 2.7.2. Algoritmo Viewer

El input del programa tiene la siguiente estructura:

./viewer -o filename.(format)

- -o: este caracter debe ser reemplazado por los siguientes valores:
  - -d: para mdl
  - -o: para off
  - -m: para m3d
  - -v: para mvm
  - -t: para vtk
  - -e: para habilitar la visualización de puentes internos (mallas volumétricas).
- filename: Es el nombre del archivo a usar.

Programa desarrollador por Claudio Lobos que permite visualizar de una forma más cómoda mallas geométricas en diferentes formatos utilizando un entorno linux o mac.

Este programa permite recibir como input una malla geométrica en los diferentes formatos nombrados anteriormente. Realizando una lectura de éste, permite recorrerlo completamente almacenando las caras de la malla en una clase llamada FaceContainer, con el cual se procede a generar un archivo equivalente al ingresado como input en formato .off, para así poder ser abierto por el programa geomview.

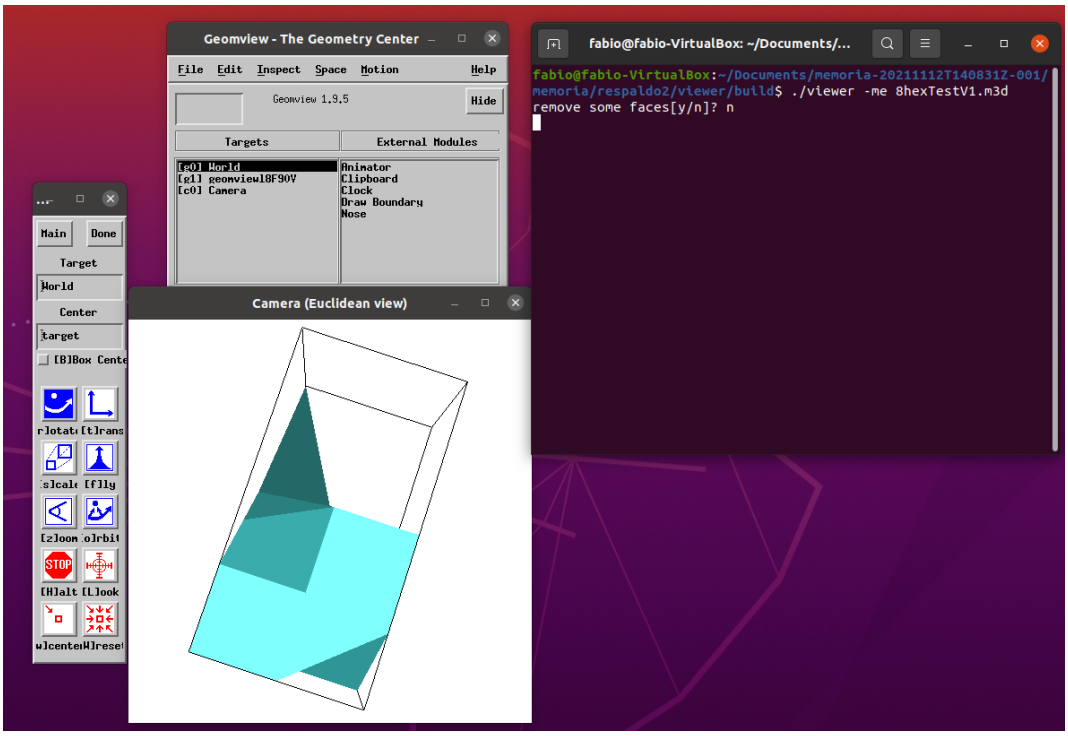


Figura 15: Screenshot Terminal ejecutando el algoritmo viewer.
Fuente: Elaboración propia.

### 2.7.3. Jens_Quality

El input del programa tiene la siguiente estructura:

./jens -option filename.m3d

- -option: puede ser reemplazado por los siguientes valores:
  - -s: Estadísticas Jacobianas Escaladas
  - -e: Estadísticas Jacobianas Elememtos Normalizados Escalados
  - -j: Lista Jens de cada elemento [Js]
  - -a: List Aspect Ratio for heach element Lista de Relación de aspecto para cada elemento
- filename: Es el nombre del archivo a usar

Programa desarrollado por Claudio Lobos que recibe como input una malla geométrica en formato .m3d, al cuál se le puede aplicar las diferentes métricas nombradas seleccionando la letra correspondiente.Para el propósito de esta memoria, se utiliza la métrica JENS para poder medir la calidad de la malla volumétrica utilizando la opción -e.

El programa entrega como output por consola la cantidad de nodos que tengan una calidad negativa y de nodos que estén en rangos de 0.5 desde 0.0333 hasta 1. Además entrega la calidad del elemento de menor valor, mayor valor y promedio de los diferentes tipos de elementos pertenecientes a la malla volumétrica.

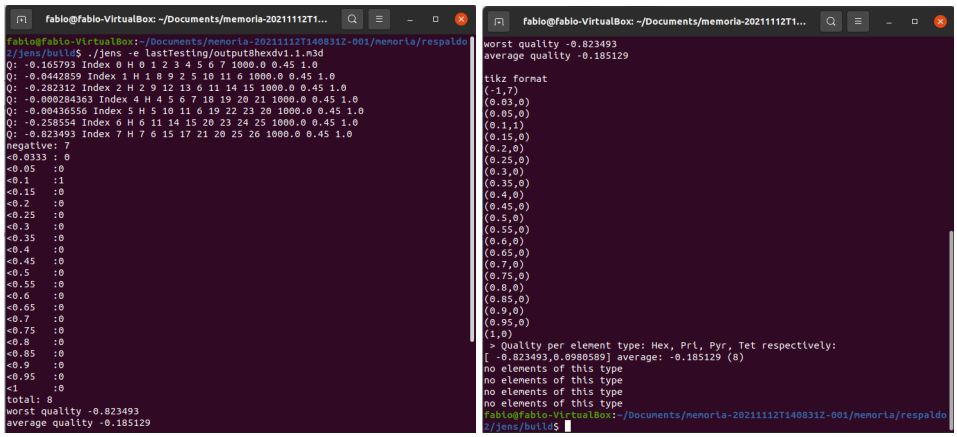


Figura 16: Screenshot Terminal ejecutando el algoritmo jens_quality.
Fuente: Elaboración propia.

## 2.8. Sistema operativo

Como los algoritmos a usar fueron programados con el lenguaje c++, es ideal trabajar con sistemas Linux o mac, optando finalmente para esta memoria por el uso de Linux.

Para esto, se utilizó del software Oracle VM Virtual Box, el cuál permite crear máquinas virtuales con diferentes sistemas operativos, optando por el uso de la distribución de Linux Ubuntu.

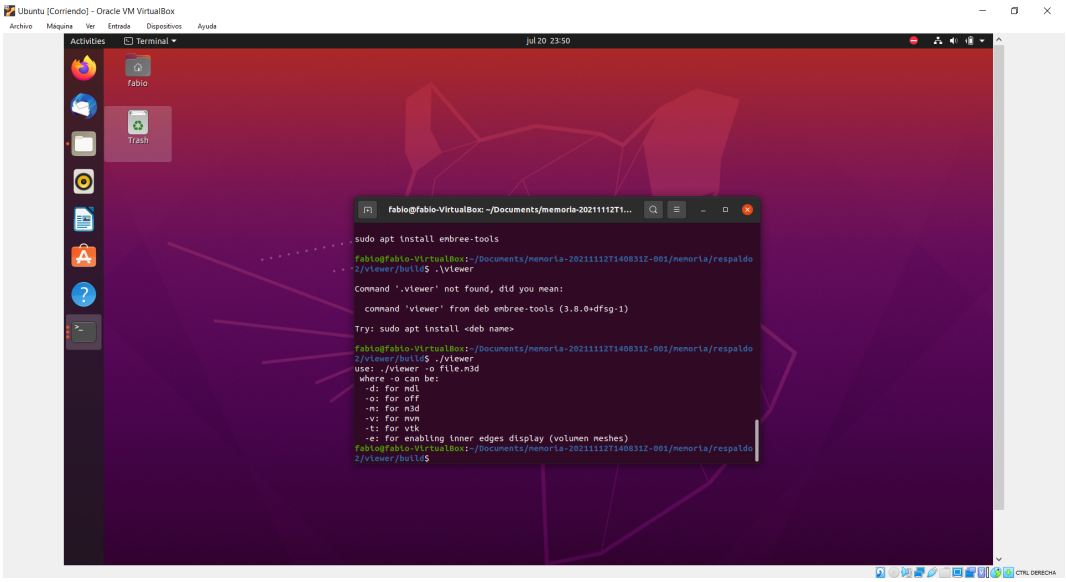


Figura 17: Screenshot VirtualBox.
Fuente: Elaboración propia.

# CAPÍTULO 3
# PROPUESTA DE SOLUCIÓN

## 3.1. PROBLEMA A RESOLVER

Para trabajar en la propuesta de solución, Se utilizará un hexaedro dividido en ocho hexaedros. Se seleccionó esta malla ya que es sencilla y fácil de deformar para poder crear diferentes situaciones.

Al medir la calidad de esta malla geométrica usando el algoritmo Jens_Quality, obtenemos un valor de 1, que es considerado como la máxima calidad posible a alcanzar, esto ya que la malla no está deformada y por ende el calculo de los jacobianos en los elementos dieron 1.

Para poder realizar pruebas, el hexaedro de la figura 8 fue modificado desplazando el nodo central hacia la parte superior, logrando atravesar la parte superficial, quedando la siguiente figura.

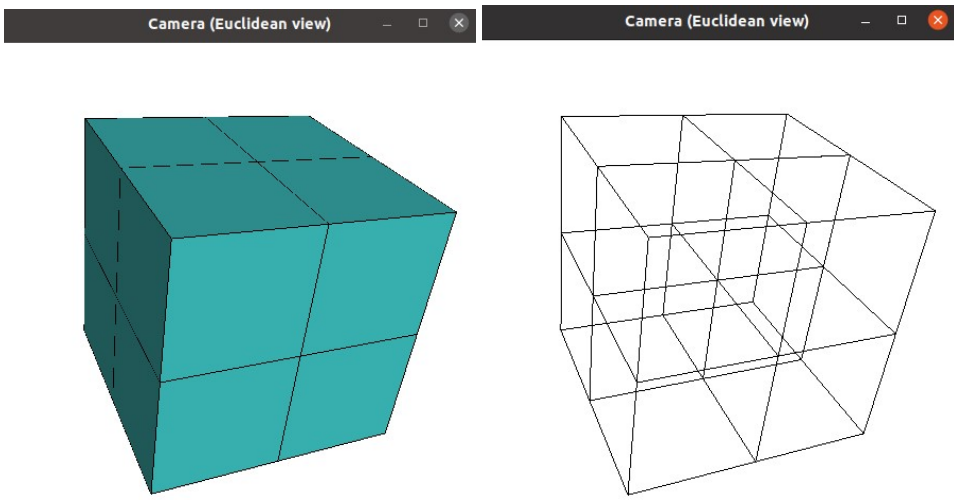


Figura 18: Hexaedro perfecto con una subdivisión de 8 hexaedros.
Fuente: Elaboración propia.

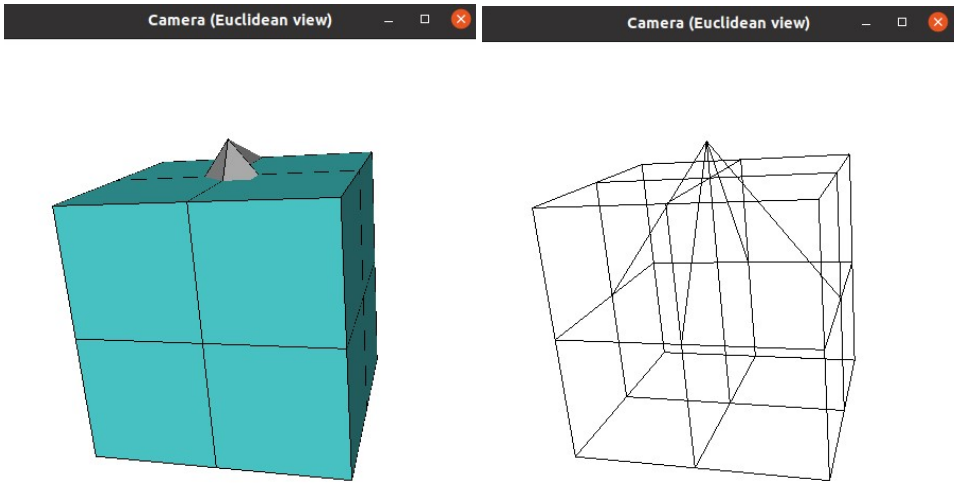


Figura 19: Hexaedro con subdivisión de 8 con el nodo central desplazado a la superficie superior.
Fuente: Elaboración propia.

La malla geométrica es ingresada al programa Jens_repair, dando el siguiente resultado con la siguiente calidad.

Al ejecutar el algoritmo en una malla volumétrica, se observa el siguiente resultado.

En la figura se puede apreciar que el problema se encuentra en la superficie de la malla.

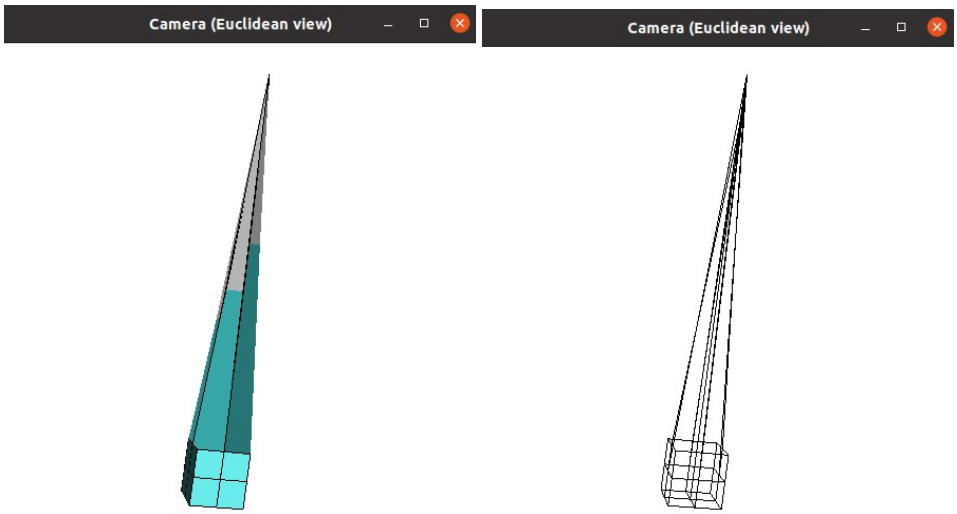


Figura 20: Hexaedro de la figura 8 luego de pasar por el programa Jens_repair.
Fuente: Elaboración propia.

## 3.2. PROPUESTA DE SOLUCIÓN

Para poder resolver el problema presentado en la figura 21, se trabajará en la modificación del algoritmo del programa Jens_Repair.

A continuación se presenta el proceso de modificación realizado al algoritmo Jens_Repair para poder llevar a cabo los objetivos propuestos en esta memoria.

### 3.2.1. Separación de nodos

El primer paso fue almacenar todos los nodos que fuesen superficiales o pertenezcan a elementos superficiales.

En la figura 22, los nodos superficiales son aquellos donde las caras fueron coloreadas de color celeste, y son considerados pertenecientes a elementos superficiales a los nodos internos que junto con 1 o más nodos superficiales forman alguno de los elementos geométricos nombrados en la sección 2.2.2.

Esta separación se logró usando la clase FaceContainer, la cuál fue sacada del algoritmo Viewer.

La clase FaceContainer es un arreglo que permite almacenar las caras de un elemento. De esta forma, nos permitirá recorrer la malla volumétrica y saber que caras son superficiales.

El primer paso para lograr arreglar la superficie de la malla fue el siguiente.

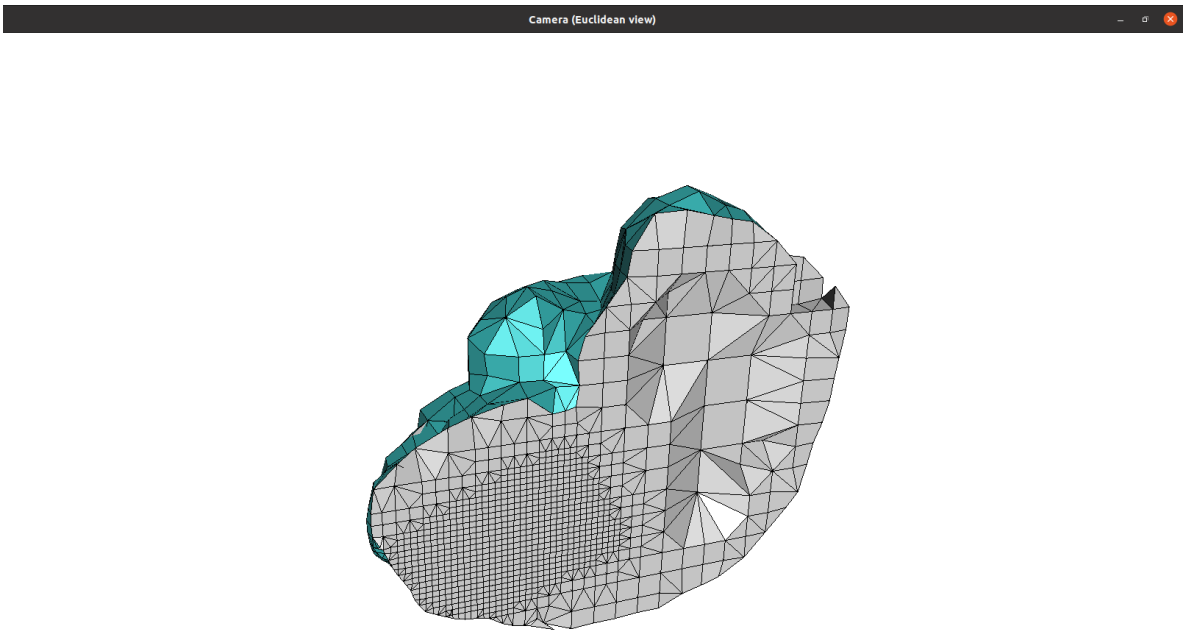


Figura 21: malla volumétrica de un cerebro cortado.
Fuente: Elavoración propia.

Se ingresaron algunos ejemplos en el programa jens_repair para verificar cuál era el problema.

Se encontró que algunos nodos se desplazaban en sentido diferente al que deberían (calculado por el JENS), esto provocaba que nodos nunca llegasen a una calidad aceptable, la cual se puede apreciar en las figuras 20 y 21, donde el nodo a mover se desplazó hacia arriba en vez de hacia abajo como debió ser.

Para poder solucionar este problema se procedió con la siguiente propuesta.

### 3.2.2. Desplazamiento contrario al calculado

Para los nodos superficiales, se modificó el código para que el desplazamiento sea siempre en el sentido contrario al calculado, esto es, cambiar el signo de los desplazamientos dx, dy y dz entregados por la métrica JENS.

Esta implementación comenzó solucionando bien los nodos superficiales, sin embargo, se presentó un segundo problema.

Los nodos internos de los elementos superficiales tenían el mismo problema que los nodos superficiales.

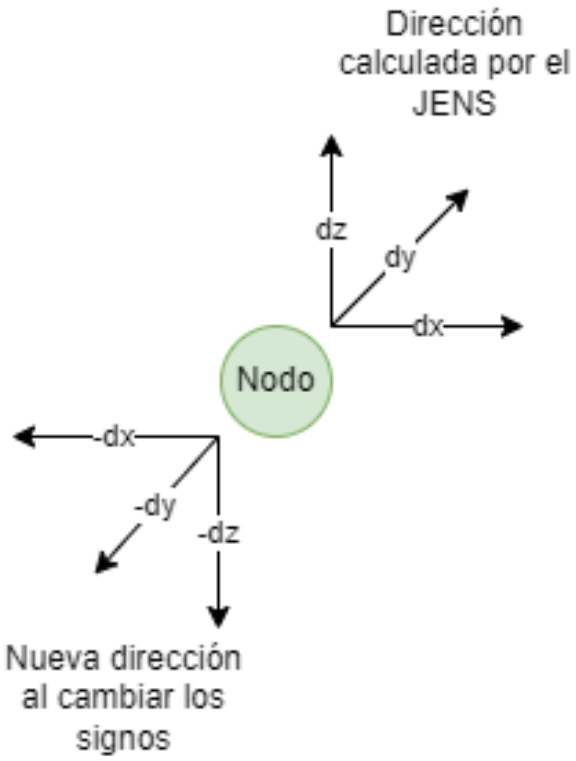


Figura 22: ejemplo de cambio de dirección de desplazamiento de un nodo.
Fuente: Elaboración propia.

Es por esto que se procede a hacer la separación de nodos superficiales, nodos internos de elementos superficiales, y los nodos internos. Quedando así tres conjuntos de nodos en total.

Se procede a aplicar el mismo procedimiento para los nodos internos de los elementos superficiales.

Al hacer esto la malla con la que se estaba trabajando logra ser validada. El siguiente paso después de validar la malla era mejorar la calidad de la malla de tal forma que se parezca lo más posible al modelo a representar.

### 3.2.3. Implementación de Cascarón

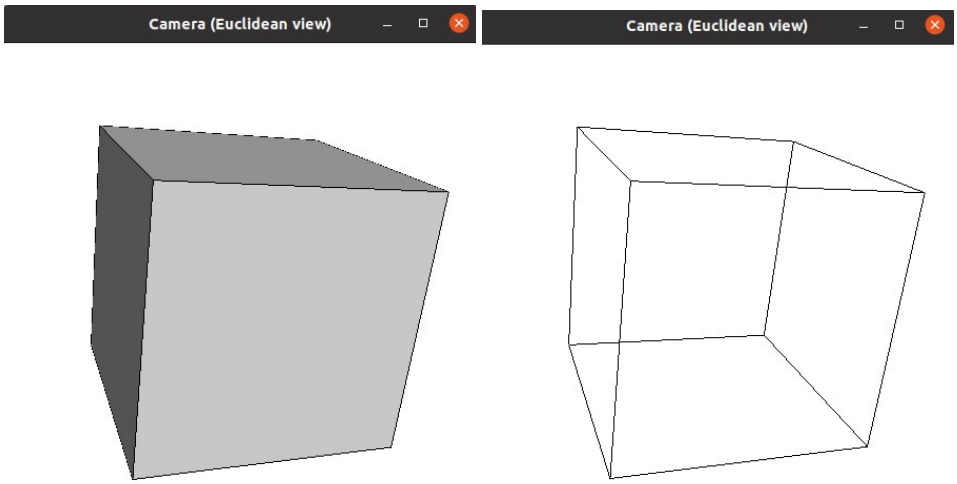


Figura 23: malla superficial de un Hexaedro perfecto.
Fuente: Elaboración propia.

Se modifica el código para que esta reciba como input una malla superficial equivalente al que se quiere llegar, la cuál definiremos como cascarón. La malla debe estar en formato mdl (malla superficial).

Para todos los nodos superficiales que fueron desplazados hasta llegar a la calidad deseada, se les aplica una proyección hacia el cascarón, desplazándolos hacia la dirección más corta hasta llegar al cascarón.

Esto se logra al implementar la función “projection()” existente dentro del código del programa.

Implementando esta modificación se logró llegar a una malla muy similar al deseado, sin embargo, al testear con otras mallas, volvió a ocurrir el problema “solucionado anteriormente”.

Se descubre que el problema no era solo en los nodos de los elementos superficiales, sino que ocurría con todos los nodos de la malla que fuesen de calidad negativa. Esto incluye tanto nodos superficiales como internos (todos), por lo que la separación de los conjuntos de nodos realizado en el paso anterior ya no es de tanta utilidad.

Se implementa el desplazamiento contrario al calculado de todos los nodos que haya que reparar.

Problema, implementar el desplazamiento contrario provocó que nodos que debían moverse correctamente, al cambiarles de signo, se desplacen en sentido erróneo, esto es, que bajan de calidad.

Se concluye que esta solución no es viable, por lo que se procede a buscar una nueva forma de desplazar los nodos.

### 3.2.4. Cambio de signo con patrón

Se intenta buscar un patrón en los nodos para ver si es posible diferenciar que nodos hay que cambiar la dirección de desplazamiento.

Se realizan las primeras combinaciones. Pero después de verificar algunas de estas, ciertas combinaciones podían entregar 2 resultados diferentes, era posible que el desplazamiento fuera correcto o incorrecto, por lo que fue imposible encontrar un patrón.

Tanto el desplazamiento contrario como el cambio de signo con patrón, fueron soluciones que no tenían una base de por qué debía hacerse así, por lo que estas soluciones quedan descartadas por la falta de fundamentos.

### 3.2.5. Desplazamiento inteligente

En esta propuesta de solución, para cada nodo a mover, se implemento un desplazamiento inteligente, esto es, se hace una prueba de todas las direcciones en las que se puede desplazar un nodo, siendo un total de nueve, quedándose con la dirección que mejora en mayor medida su calidad.

A continuación se presenta un pseudocódigo de esta implementación.

```
for nodo in malla do
    QualitiesArray =[]
    desplazar nodo en dirección dx, dy, dz
    newQuality = JensQuality(nodo)
    agregar newQuality a QualitiesArray
    desplazar nodo a su posición original

    desplazar nodo en dirección -dx, dy, dz
    newQuality = JensQuality(nodo)
    agregar newQuality a QualitiesArray
    desplazar nodo a su posición original

    desplazar nodo en dirección dx, -dy, dz
    newQuality = JensQuality(nodo)
    agregar newQuality a QualitiesArray
    desplazar nodo a su posición original

    desplazar nodo en dirección dx, dy, -dz
    newQuality = JensQuality(nodo)
    agregar newQuality a QualitiesArray
    desplazar nodo a su posición original
```

```
    implementamos lo mismo para las direcciones faltantes...
    ...
    ...
    ...
    ...
    MaxQuality=0
    for Quality in QualitiesArray do
        if Quality >MaxQuality then
            MaxQuality=Quality
        end if
    end for
end for
```

Esta implementación, si bien desplazaba el nodo a donde mejoraba de forma máxima su calidad, también provocaba que otros nodos perdieran calidad e incluso a tal punto de invalidarlos, quedando en un bucle infinito de no ser por el número máximo de iteraciones. Además, la dirección a la que se desplazaba a veces no era la ideal para que la malla se pareciese a la original.

Esta propuesta no logró solucionar el problema, además, era muy costoso en tiempo, dado que su tiempo de ejecución iba creciendo al tener nueve veces el cálculo del jacobiano por nodo en la malla, además, para cada nodo se itera dentro de su arreglo de calidades, siendo un proceso $for$ dentro de otro $for$. Dado lo complejo del algoritmo, se podría representar aproximadamente como $O(n^5)$ por lo que es descartada, sin embargo, al provocar problemas a los nodos cercanos donde se desplazaba el nodo, fue útil para entender que no se puede mejorar una malla completamente mejorando uno a uno los nodos, ya que al validar un nodo, puede invalidar a otros.

### 3.2.6. Cambio Proceso iterativo

Para intentar corregir la invalidación de unos nodos al desplazar otros, se propuso cambia el proceso iterativo del código. Esta nueva propuesta consta de desplazar cada nodo una sola vez iterativamente para que de a poco despejen el espacio para los nodos que se desplazan a esa dirección.

A continuación se presenta un pseudocódigo para entender la comparativa entre los método de iteraciones antes y después.

Proceso iterativo inicial:

```
for nodo in malla do
    newQuality = JensQuality(nodo)
    while newQuality sea menor a la ingresada do
        desplazar nodo
```

```
        newQuality = JensQuality(nodo)
    end while
end for
```

Proceso iterativo final:

```
for nodo in malla do
    newQuality = JensQuality(nodo)
    if newQuality menor a la ingresada then
        desplazar nodo
    end if
end for
```

Se entiende que esta implementación es importante por el análisis realizado anteriormente, por lo que se mantendrá este cambio.

En esta parte se empieza a analizar que quizás no sea lo ideal mejorar la calidad del nodo al desplazarlo de forma individual, sino que esto a su vez, debe mejorar la calidad de los nodos vecinos.

### 3.2.7. Enfoque de sistema

Inicialmente, el proceso iterativo para mejorar la calidad de los nodos de la malla se realizaba calculando la calidad solamente del nodo a desplazar, iterando en este hasta que lograse superar la calidad mínima ingresada antes de seguir con el siguiente nodo.

Esta nueva propuesta busca que para cada nodo invalido a mover, también se calculen las calidades de los nodos vecinos, calculando un promedio de estos. Buscamos mejorar el promedio del nodo y sus vecinos en vez de la calidad del nodo individualmente, además, como existe la posibilidad de que el promedio mejore, pero la calidad del nodo desplazado no, también se toma en cuenta la calidad de este de forma independiente.

A continuación se presenta un pseudocódigo de la nueva forma de calcular el promedio de la calidad de los nodos usando la métrica JENS:

```
sum=0
sum= sum + calcular calidad del nodo
for nodovecino in nodo.vecinos do
    sum= sum + calcular calidad de nodovecino
end for
sum=sum/largo(nodo.vecinos)
newQuality= sum
```

Esta implementación teóricamente es correcta, pero en la práctica no logra solucionar el problema.

Este método se testea tanto en el proceso de validación como en el mejoramiento de calidad.

Logra mover de forma adecuada a nodos que tienen calidad tanto positiva como negativa.

Se descubre que el problema surge cuando se quiere mover un nodo de calidad negativa que está conectado a otro nodo negativo. Cuando esto ocurre, la dirección calculada por el jens permite mejorar la calidad, pero es incapaz de validar el nodo, esto ya que el nodo debiese ser desplazado en sentido contrario. sin embargo, si desplazamos el nodo hacia la dirección correcta, este empieza a disminuir aún más su calidad, pero el nodo llega a una posición en la cuál su calidad se vuelve positiva, es por esto que se requiere abordar de una forma distinta.

### 3.2.8. Mover solo uno de los nodos negativos conectados

Se implementa un método que permite conocer los nodos vecinos conectados al nodo negativo y ver cuál de estos es también negativo.

Se realizan dos pruebas:

- Mover el nodo más negativo de los nodos conectados.

  El resultado permitió validar ambos nodos negativos, sin embargo, esto provocó que se formaran otros subconjuntos de nodos negativos conectados, por lo que no logró validar la malla.

- Mover el nodo menos negativo de los nodos conectados.

  Se produjo el problema nombrado anteriormente, el nodo se desplaza hacia la dirección donde mejora su calidad, pero no logra validarse.

### 3.2.9. Intercambio de posición de nodos vecinos negativos

Esta implementación busca tomar dos nodos negativos conectados e intercambiar sus posiciones, esta implementación logró validar la malla utilizada, sin embargo, su funcionamiento se vuelve complejo al haber mas de dos nodos negativos conectados.

### 3.2.10. Desplazamientos de nodos negativos en 2 direcciones

Como se nombró con anterioridad, cuando dos nodos negativos están conectados, el cálculo de la dirección usando el jens no nos permite resolver estas situaciones, se implementa un método para que los nodos negativos prueben en la dirección calculada por el jens y en la dirección opuesta a este, con el fin de que los nodos negativos tengan más posibilidades de ser reparados.

### 3.2.11. Implementación en mejoramiento de calidad

Finalmente, se implementa el uso del cascarón, el cambio del proceso iterativo y el cálculo de calidad con un enfoque de sistema para intentar mejorar la calidad de la malla volumétrica.

Se logra mejorar la calidad considerablemente, mostrando un resultado muy cercano a lo que se quiere llegar.

A continuación se muestra visualmente el modelo a evaluar:

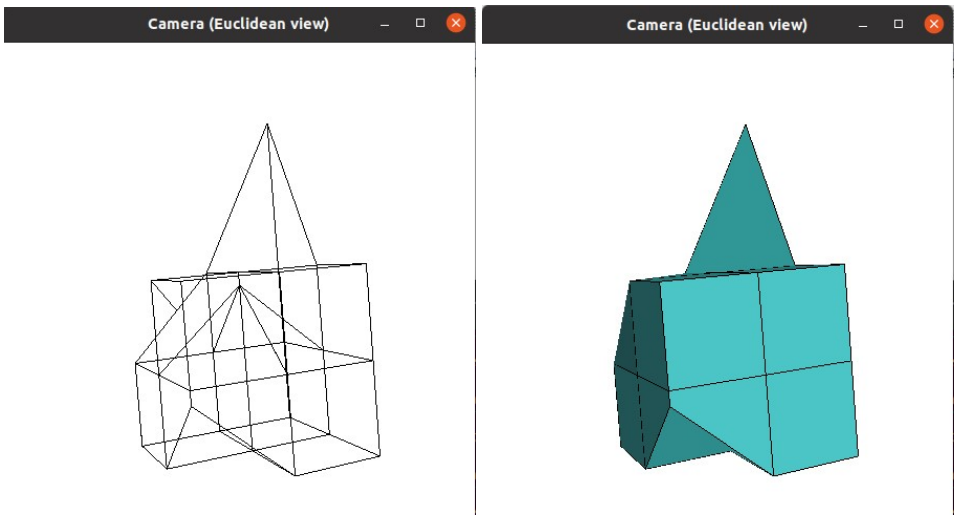


Figura 24: Hexaedro con una división de 8 modificado.
Fuente: Elaboración propia.

Se realiza la prueba sin usar la proyección al cascarón. Dando el siguiente resultado:

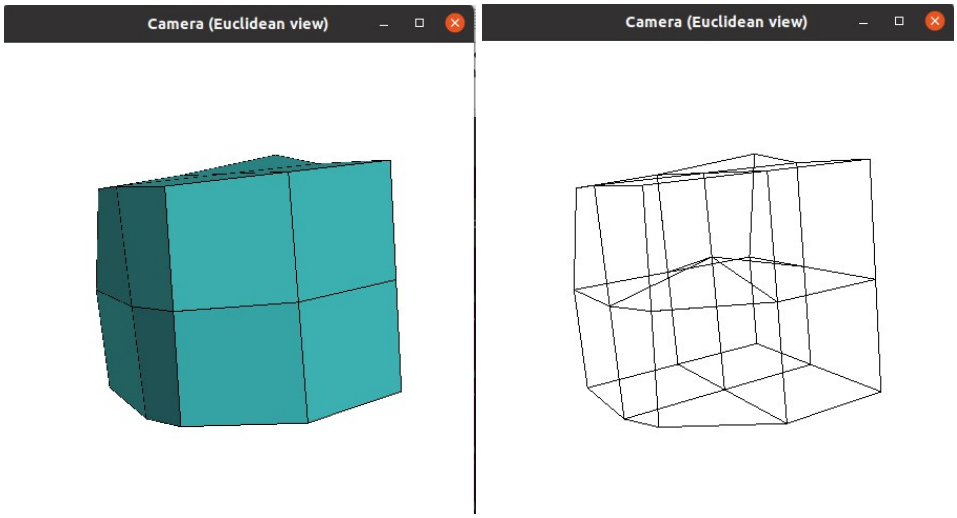


Figura 25: Hexaedro de la figura 25 luego de pasar por el programa Jens_repair sin usar proyección.

Fuente: Elaboración propia.

Se realiza la prueba usando la proyección al cascarón. Resultado con proyección

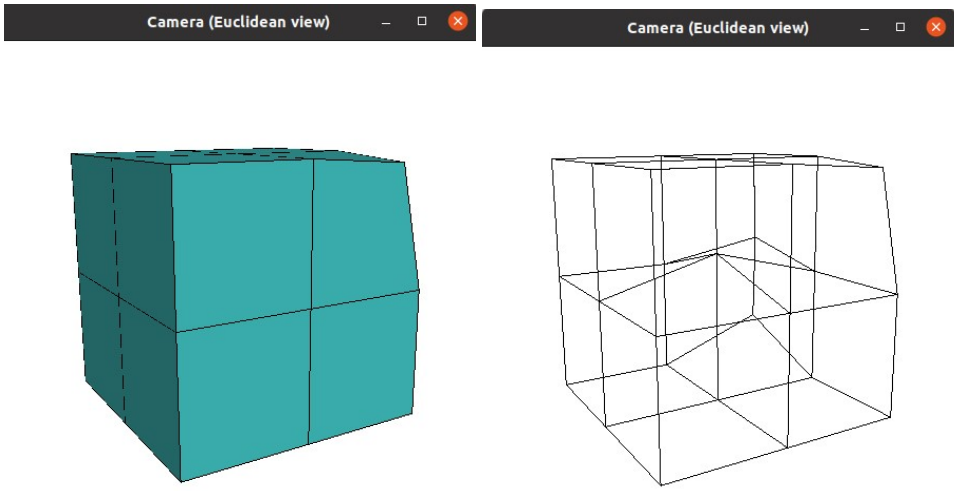


Figura 26: Hexaedro de la figura 12 luego de pasar por el programa Jens_repair usando proyección.

Fuente: Elaboración propia.

Se realiza la prueba usando la proyección al cascarón, además de utilizar un threshold alto.

Como la implementación logra un resultado considerablemente bueno, por lo que se procede a concluir la propuesta de solución.

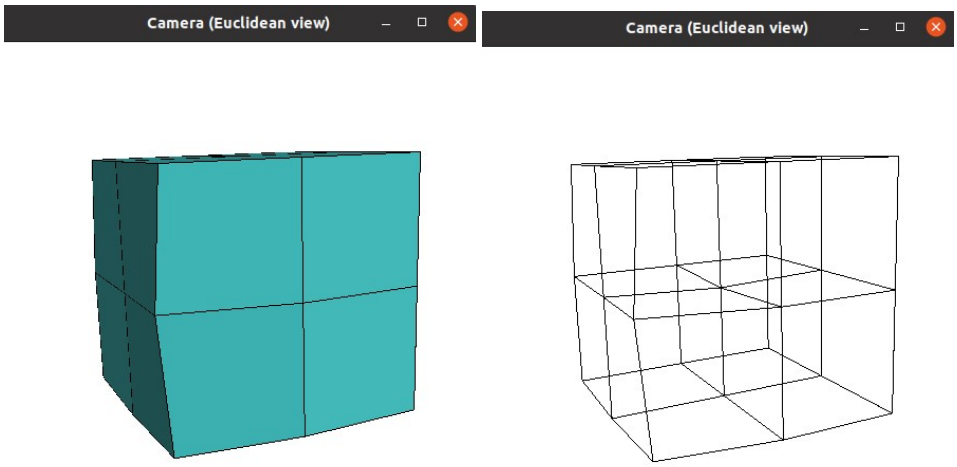


Figura 27: Hexaedro de la figura 12 luego de pasar por el programa Jens_repair usando proyección y un threshold alto.

Fuente: Elaboración propia.

# CAPÍTULO 4
# VALIDACIÓN DE LA SOLUCIÓN

Para poder validar la solución propuesta, en esta sección se ofrecerá una muestra de diferentes mallas volumétricas con diferentes cantidades de subdivisiones(elementos) utilizadas para hacer las pruebas necesarias para verificar que la propuesta de solución es válida.

## 4.1. Mallas volumétricas usadas

Mallas utilizadas inicio y final visualizadas usando GeomView:

Las pruebas se partieron usando una malla equivalente al elemento más simple posible, un hexaedro sin subdivisiones, con una deformación sencilla.

Input: Hex1.m3d

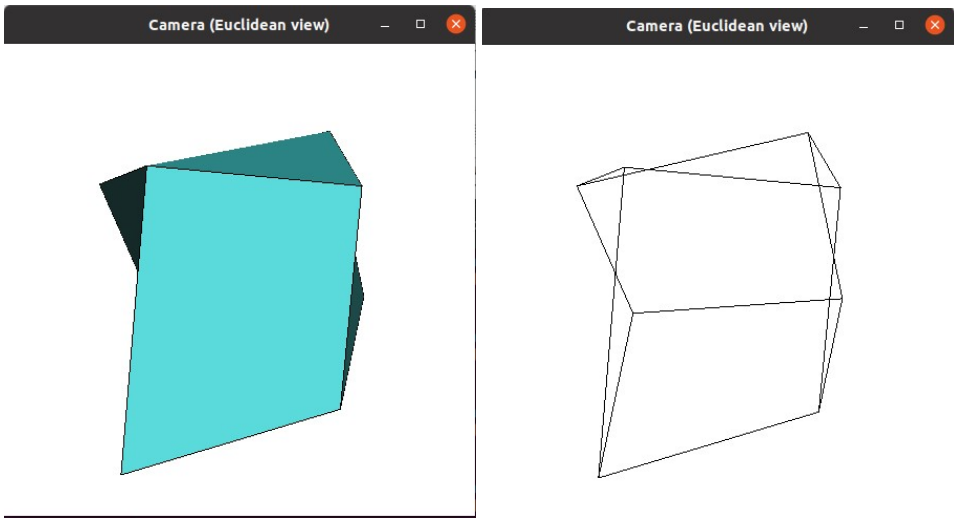


Figura 28: Hexaedro modificado 1.
Fuente: Elaboración propia.

Output: OHex1.m3d

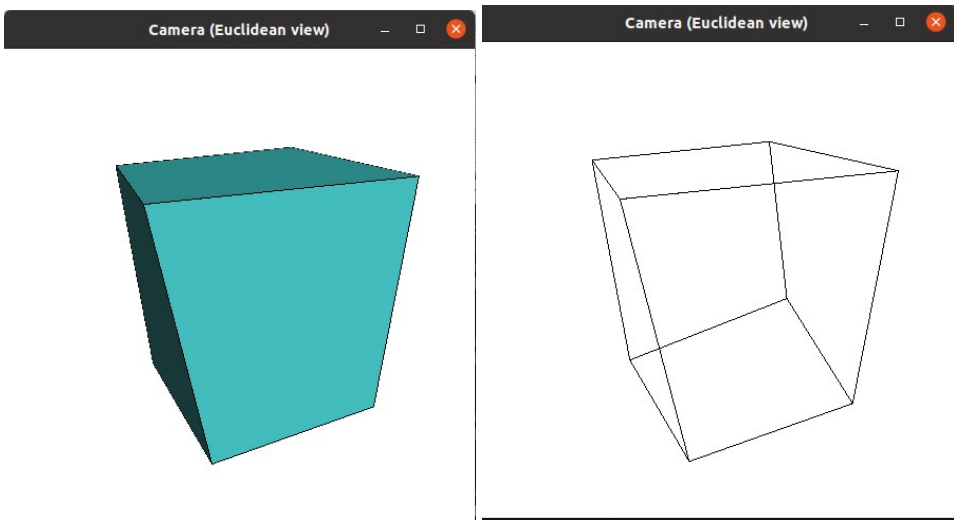


Figura 29: Resultado hexaedro modificado 1.
Fuente: Elaboración propia.

La siguiente malla volumétrica es equivalente a la anterior, pero con una deformación mayor.

Input: Hex2.m3d

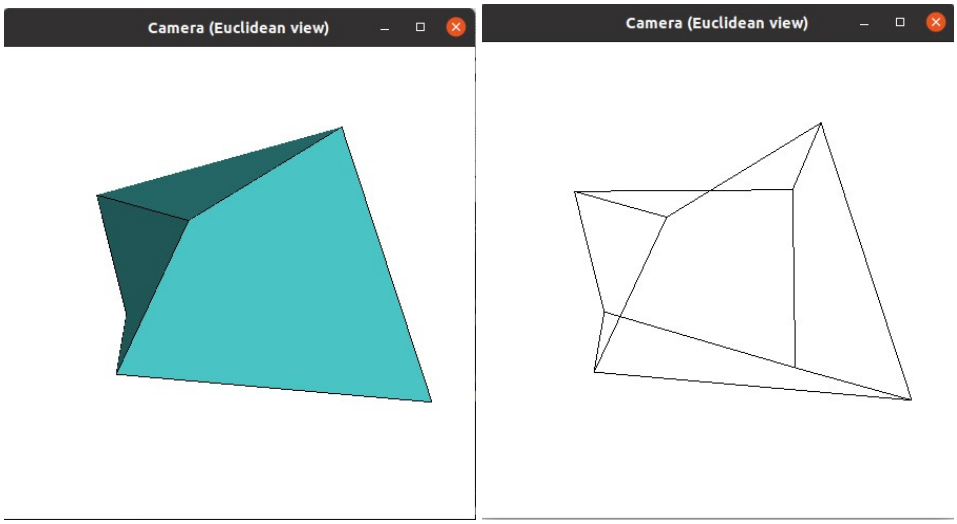


Figura 30: Hexaedro modificado 2.
Fuente: Elaboración propia.

Output: OHex2.m3d

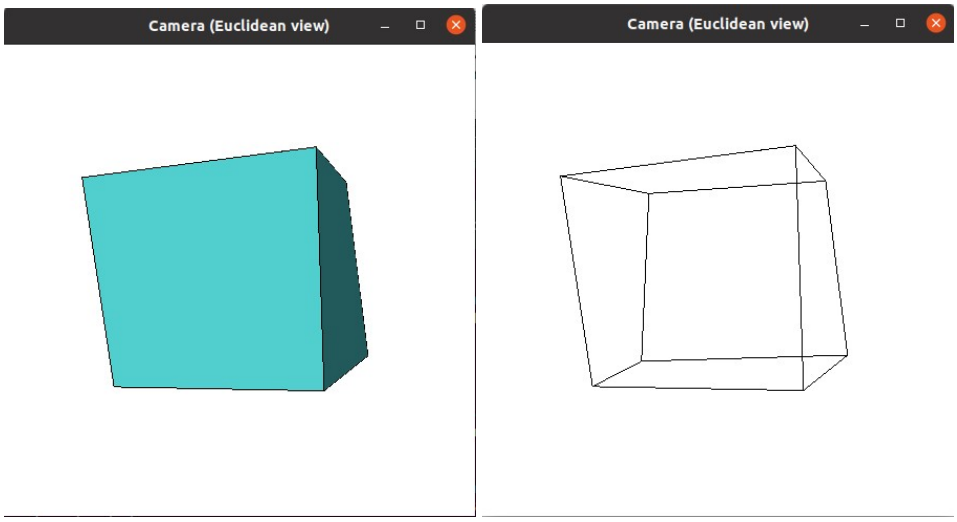


Figura 31: Resultado hexaedro modificado 2.
Fuente: Elaboración propia.

La siguiente malla volumétrica es un hexaedro con una subdivisión de 8 hexaedros con una primera deformación.

Input: 8Hex1.m3d

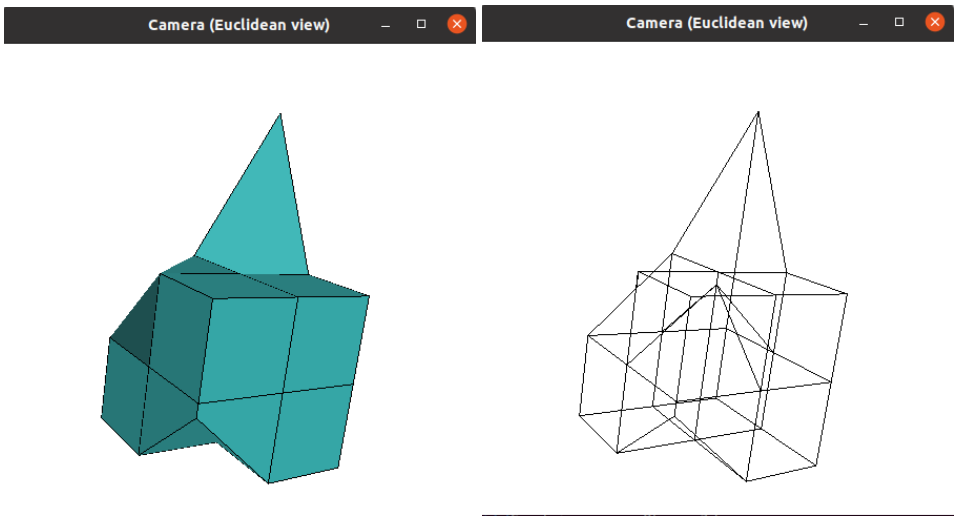


Figura 32: Resultado hexaedro modificado 2.
Fuente: Elaboración propia.

Output: O8Hex1.m3d

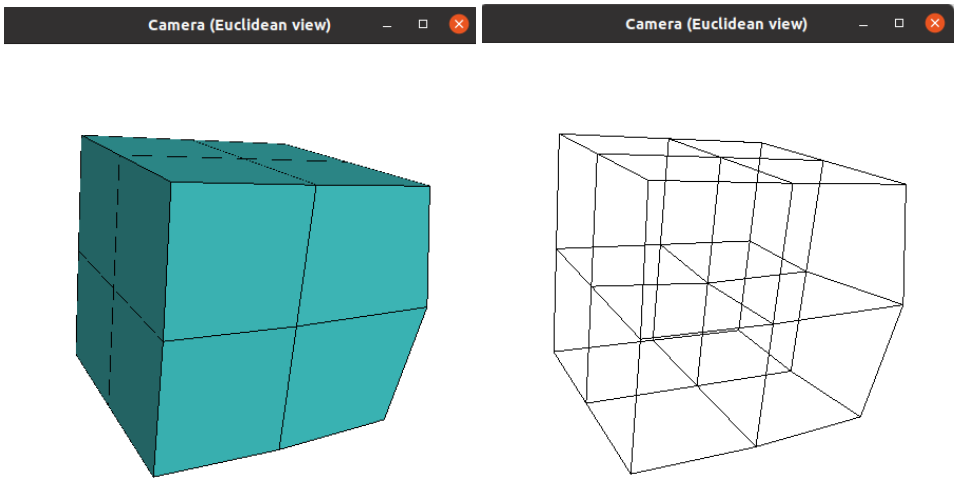


Figura 33: Resultado hexaedro modificado 2.
Fuente: Elaboración propia.

La siguiente malla volumétrica es equivalente a la anterior pero con una segunda deformación.

Input: 8Hex2.m3d

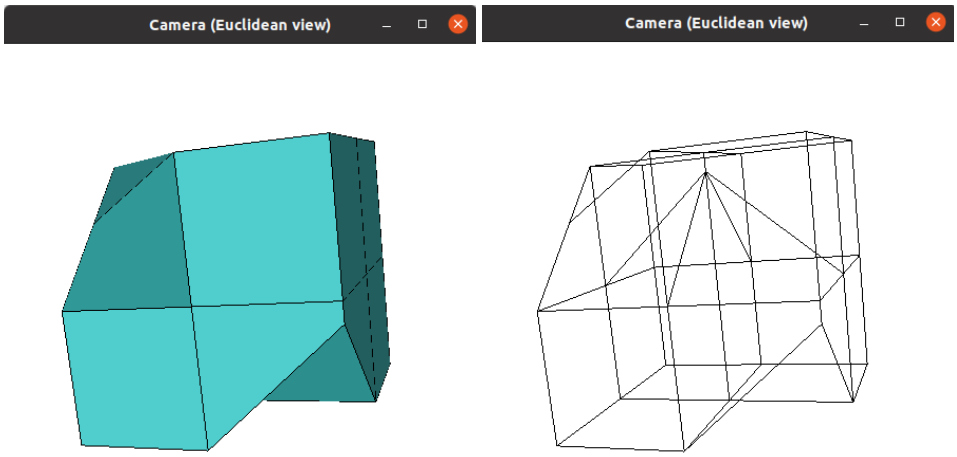


Figura 34: Resultado hexaedro modificado 2.
Fuente: Elaboración propia.

Output: O8Hex2.m3d

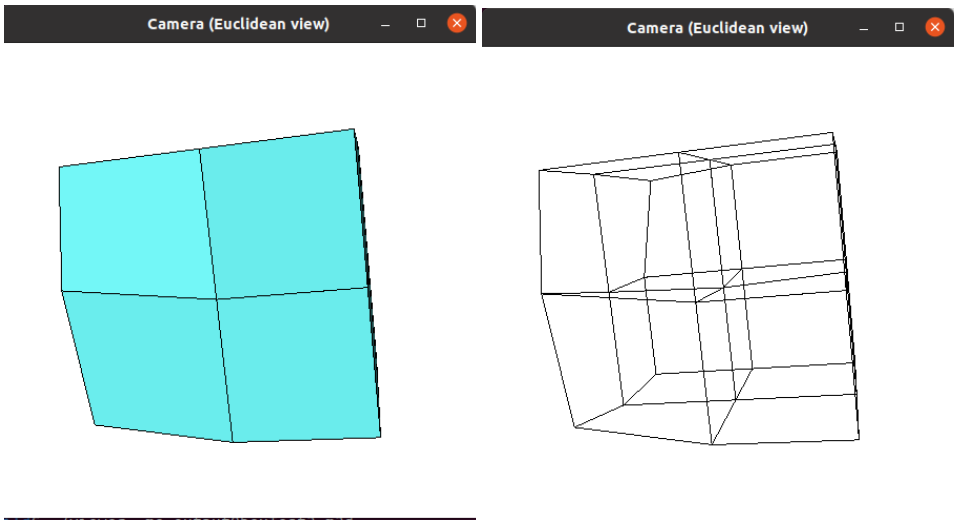


Figura 35: Resultado hexaedro modificado 2.
Fuente: Elaboración propia.

Si bien el aspecto visual se ve claramente una mejora en las mallas testeadas, es de mayor utilidad analizar las calidades obtenidas por cada una de las mallas geométricas, por lo que se ofrece una tabla con los valores obtenidos a través del algoritmo Jens_Quality.

Tabla 2: Calidades de las mallas utilizadas antes y después de la mejora de calidad. Fuente: Elaboración Propia.

| Modelo de entrada | Calidad mínima | Calidad máxima | Calidad promedio | Modelo de salida | Calidad mínima | Calidad máxima | Calidad promedio |
|---|---|---|---|---|---|---|---|
| Hex1 | 0.856867 | 0.856867 | 0.856867 | OHex1 | 0.951344 | 0.951344 | 0.951344 |
| Hex2 | 0.016128 | 0.016128 | 0.016128 | OHex2 | 0.921803 | 0.921803 | 0.921803 |
| 8Hex1 | 0 | 0.576471 | 0.460449 | O8Hex1 | 0.949149 | 0.998167 | 0.989505 |
| 8Hex2 | 0 | 0.576471 | 0.504412 | O8Hex2 | 0.945743 | 0.998167 | 0.983655 |

Para poder testear como el algoritmo maneja las mallas con nodos negativos, se procedió a crear un programa en python llamado meshGenerator, el cuál recibe como input una malla en formato m3d y genera nuevas mallas con la misma topología, pero donde cada nodo ha sido desplazado siguiendo un vector aleatorio.

Para esta prueba se generaron 50 mallas geométricas a partir de un hexaedro con una subdivisión de 8. Cada malla a lo menos tiene 3 nodos negativos.

Para testear las mallas se realizó un script en python el cuál ejecuta el programa jens_repair con todas las mallas generadas con el meshGenerator.

Utilizando una iteración máxima de 1000 y un threshold de 0.5, obtenemos los siguientes resultados de las 50 mallas:

- 34 mallas lograron ser reparadas y llegaron al threshold de 0.5.
- 6 mallas lograron la validación pero no llegaron al threshold de 0.5
- 10 mallas no lograron reparar los nodos negativos.

Tabla 3: Mallas de prueba parte 1.
Fuente: Elaboración Propia.

| Modelo de entrada | Cantidad de elementos negativos | logró validarse | logro llegar al threshold |
|---|---|---|---|
| eightHexInvModify0 | 4 | si | si |
| eightHexInvModify1 | 6 | si | si |
| eightHexInvModify2 | 4 | si | si |
| eightHexInvModify3 | 5 | si | si |
| eightHexInvModify4 | 4 | si | si |
| eightHexInvModify5 | 6 | no | no |
| eightHexInvModify6 | 4 | no | no |
| eightHexInvModify7 | 4 | si | si |
| eightHexInvModify8 | 4 | si | si |
| eightHexInvModify9 | 4 | si | si |
| eightHexInvModify10 | 6 | si | si |
| eightHexInvModify11 | 5 | si | si |
| eightHexInvModify12 | 4 | si | si |
| eightHexInvModify13 | 4 | si | no |
| eightHexInvModify14 | 5 | no | no |
| eightHexInvModify15 | 5 | si | si |
| eightHexInvModify16 | 7 | no | no |
| eightHexInvModify17 | 7 | si | no |
| eightHexInvModify18 | 6 | si | si |
| eightHexInvModify19 | 6 | si | no |
| eightHexInvModify20 | 5 | si | si |
| eightHexInvModify21 | 6 | no | no |
| eightHexInvModify22 | 5 | si | si |
| eightHexInvModify23 | 4 | si | si |
| eightHexInvModify24 | 4 | si | si |

Tabla 4: Mallas de prueba parte 2.
Fuente: Elaboración Propia.

| Modelo de entrada | Cantidad de elementos negativos | logró validarse | logro llegar al threshold |
|---|---|---|---|
| eightHexInvModify25 | 3 | si | si |
| eightHexInvModify26 | 4 | si | si |
| eightHexInvModify27 | 5 | si | no |
| eightHexInvModify28 | 5 | si | si |
| eightHexInvModify29 | 4 | no | no |
| eightHexInvModify30 | 5 | si | si |
| eightHexInvModify31 | 4 | si | si |
| eightHexInvModify32 | 6 | si | si |
| eightHexInvModify33 | 4 | no | no |
| eightHexInvModify34 | 5 | si | si |
| eightHexInvModify35 | 6 | si | si |
| eightHexInvModify36 | 4 | si | si |
| eightHexInvModify37 | 5 | si | si |
| eightHexInvModify38 | 4 | si | si |
| eightHexInvModify39 | 4 | si | si |
| eightHexInvModify40 | 3 | no | no |
| eightHexInvModify41 | 4 | no | no |
| eightHexInvModify42 | 5 | si | si |
| eightHexInvModify43 | 4 | si | si |
| eightHexInvModify44 | 5 | si | si |
| eightHexInvModify45 | 4 | no | no |
| eightHexInvModify46 | 4 | si | no |
| eightHexInvModify47 | 4 | si | si |
| eightHexInvModify48 | 5 | si | si |
| eightHexInvModify49 | 4 | si | si |

Para la prueba se utilizó el método planteado en la sección 3.2.10.

Acá se muestran visualmente el input y output respectivamente de las mallas que no lograron ser resueltas Y los nodos que no lograron ser reparados.

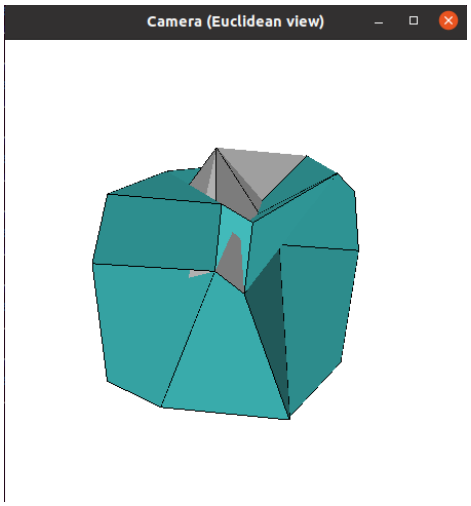


Figura 36: eightHexInvModify5

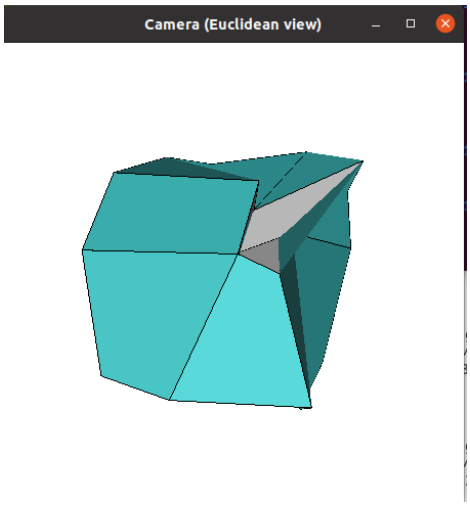


Figura 37: Output eightHexInvModify5

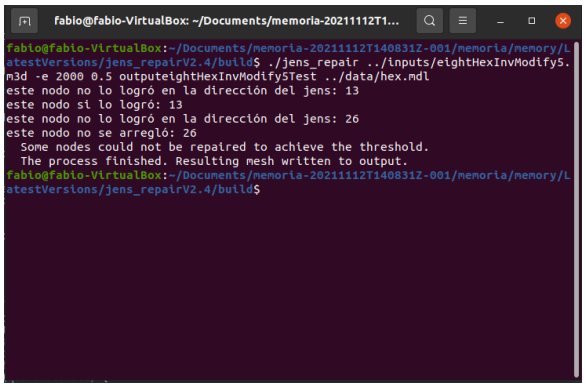


Figura 38: Ejecución jens_repair con la malla eightHexInvModify5

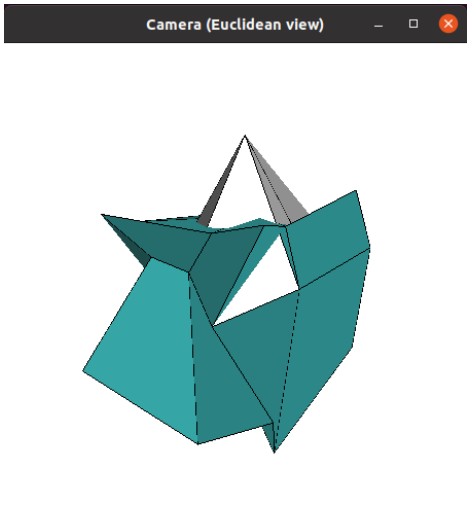


Figura 39: eightHexInvModify6

Figura 40: Output eightHexInvModify6

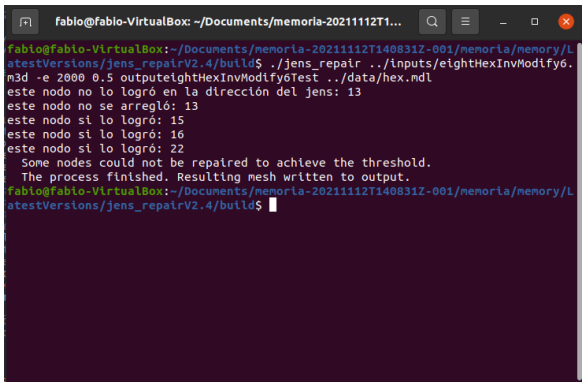


Figura 41: Ejecución jens_repair con la malla eightHexInvModify6

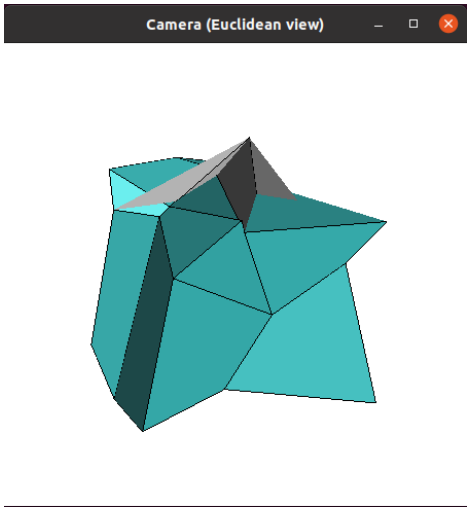


Figura 42: eightHexInvModify14

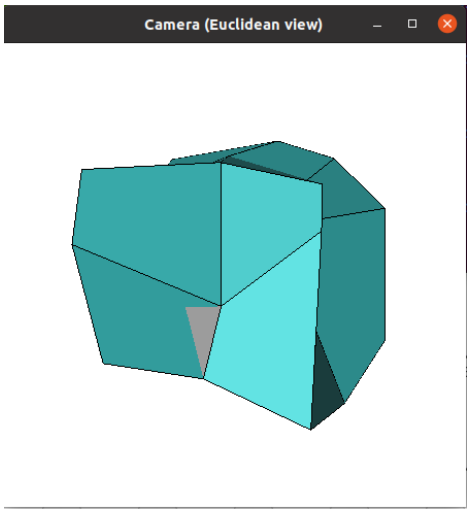


Figura 43: Output eightHexInvModify14

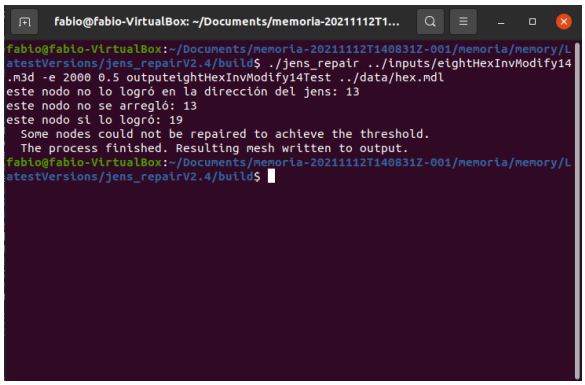


Figura 44: Ejecución jens_repair con la malla eightHexInvModify14

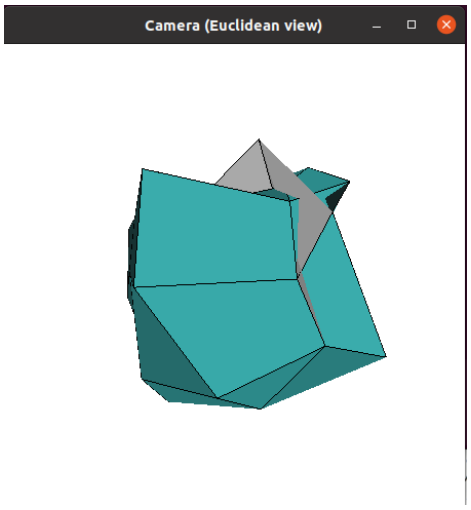


Figura 45: eightHexInvModify16

Figura 46: Output eightHexInvModify16

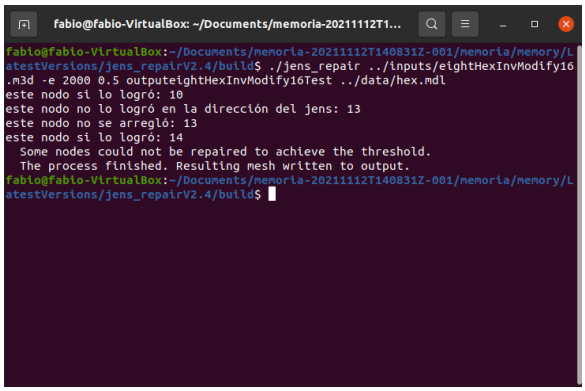


Figura 47: Ejecución jens_repair con la malla eightHexInvModify16

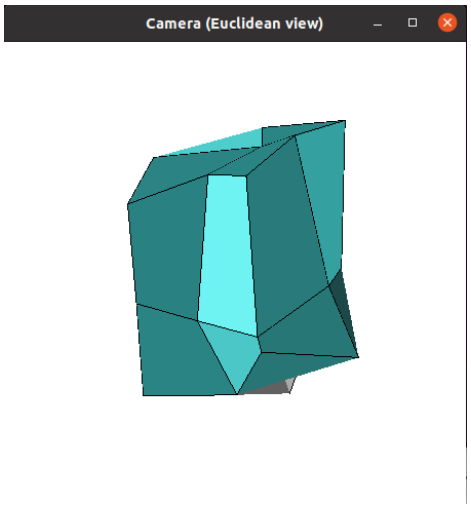


Figura 48: eightHexInvModify21

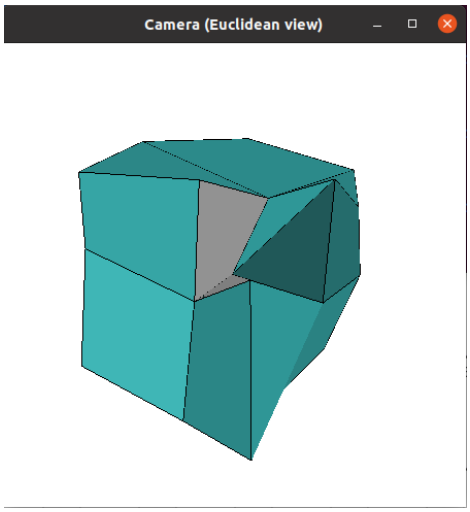


Figura 49: Output eightHexInvModify21

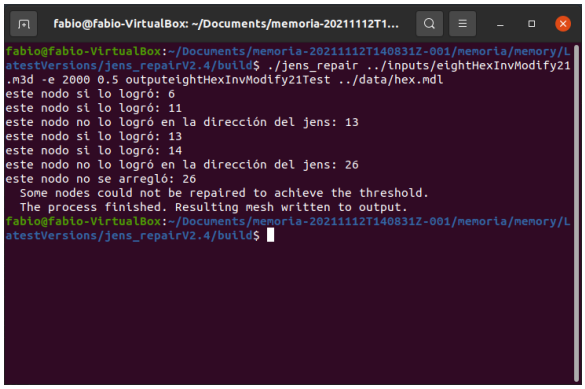


Figura 50: Ejecución jens_repair con la malla eightHexInvModify21

Figura 51: eightHexInvModify29

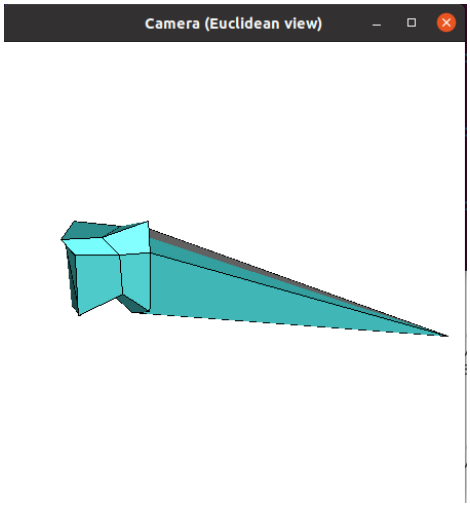


Figura 52: Output eightHexInvModify29

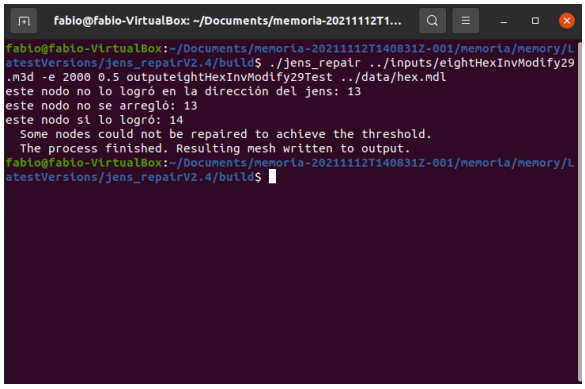


Figura 53: Ejecución jens_repair con la malla eightHexInvModify29

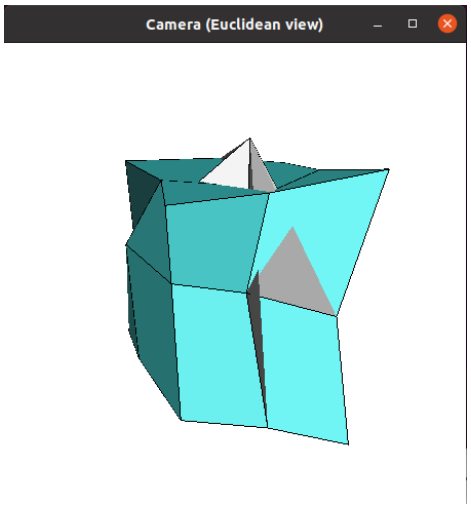


Figura 54: eightHexInvModify33

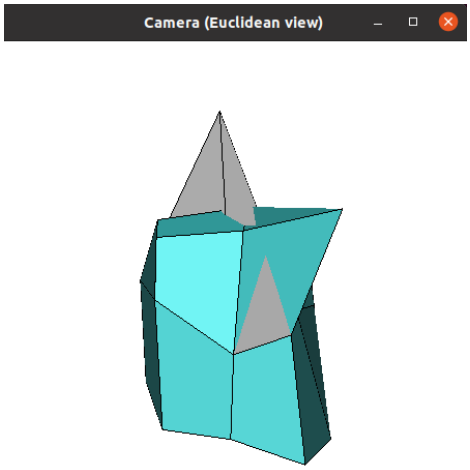


Figura 55: Output eightHexInvModify33

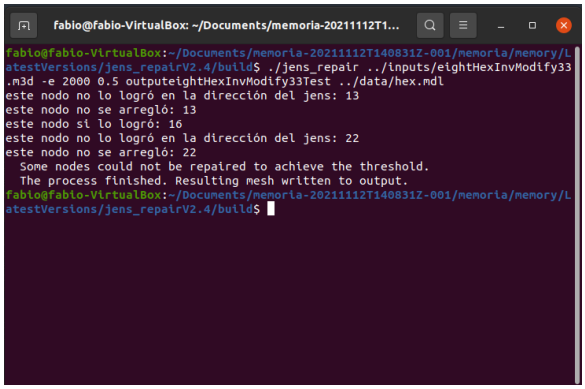


Figura 56: Ejecución jens_repair con la malla eightHexInvModify33

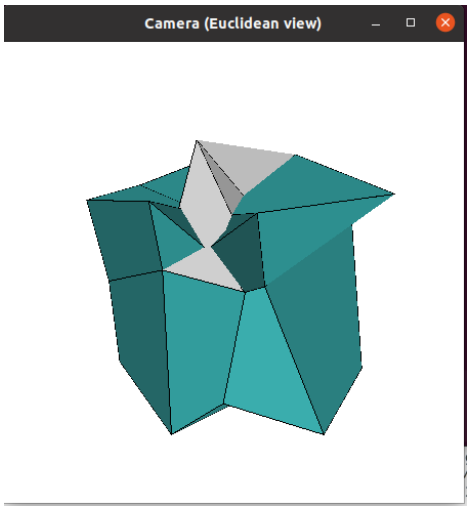


Figura 57: eightHexInvModify40

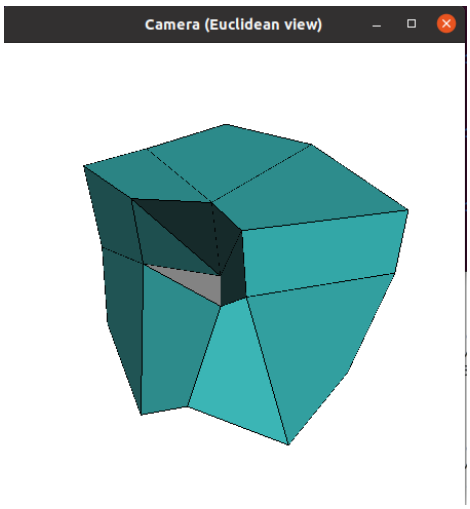


Figura 58: Output eightHexInvModify40

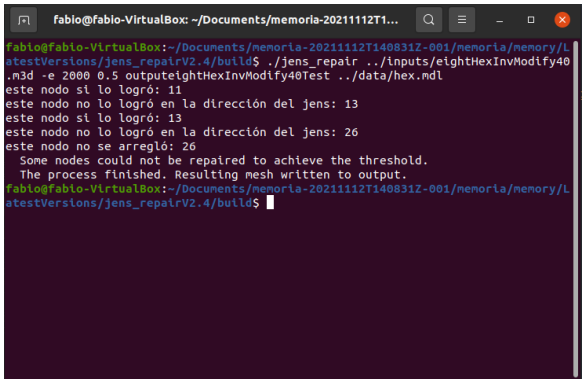


Figura 59: Ejecución jens_repair con la malla eightHexInvModify40

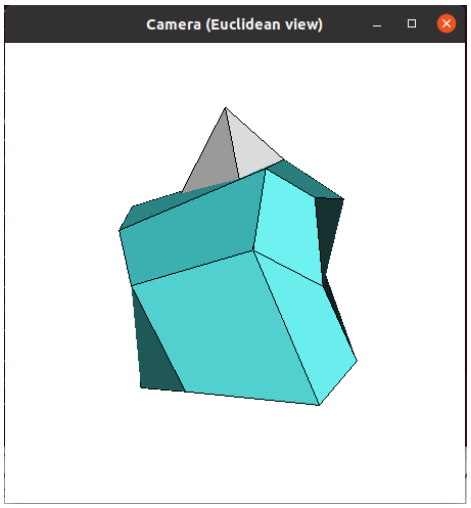


Figura 60: eightHexInvModify41

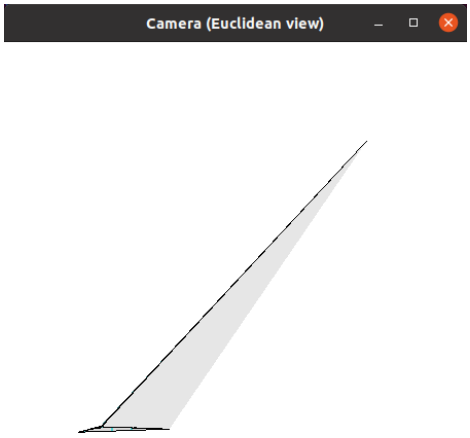


Figura 61: Output eightHexInvModify41

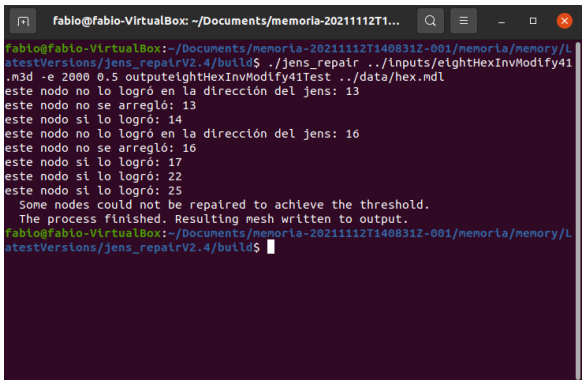


Figura 62: Ejecución jens_repair con la malla eightHexInvModify41

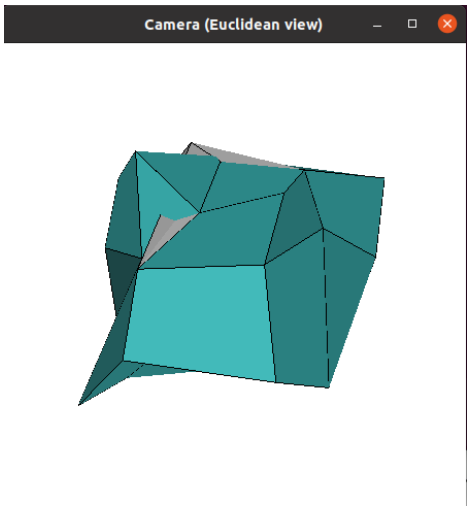


Figura 63: eightHexInvModify45

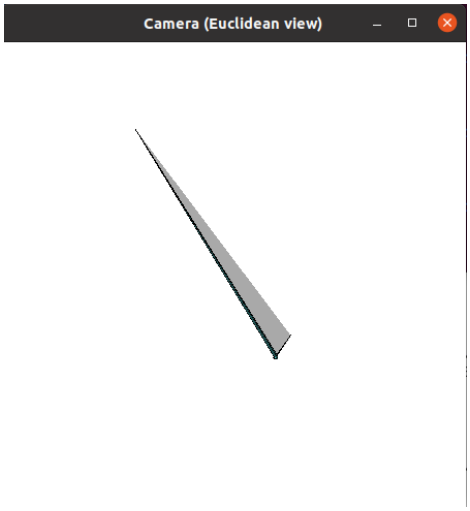


Figura 64: Output eightHexInvModify45

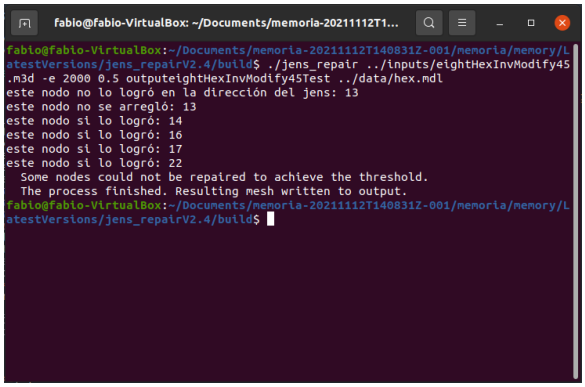


Figura 65: Ejecución jens_repair con la malla eightHexInvModify45

Tras analizar los nodos que no lograron corregirse en las mallas mostradas y realizar pruebas puntuales con estas. Se listan las siguientes causas.

- mover nodos negativos antes del nodo negativo que se requiere validar, puede provocar que este no logre validarse.
- la dirección invertida del jens solo logra validar el nodo negativo si la dirección de desplazamiento calculado por el jens es más cercano a un eje de coordenadas. De otra forma la dirección es erronea.

# CAPÍTULO 5
# CONCLUSIONES

## 5.1. Conclusiones generales

### 5.1.1. Métricas

La métrica JENS fue creada específicamente para poder solucionar el problema de no poder mejorar la calidad de mallas volumétricas formadas por los diferentes tipos de elementos geométricos, para los cuales la métrica Js y Jr no contemplaban o eran incapaces de trabajar con estos. El uso de la métrica JENS permitirá un gran avance en el área de la generación de mallas volumétricas en el futuro, ya que de esta forma se permitirá tener mallas que tengan buena calidad y además mantengan la forma del objeto que se representa y así poder trabajar más óptimamente en el área de la mecánica y biología.

### 5.1.2. Desarrollo

El proceso de desarrollo de esta memoria requirió de mucho esfuerzo y tiempo, además de un proceso de desarrollo iterativo, esto ya que el problema a resolver, que inicialmente se creía era solo superficial, terminó siendo provocado por factores más internos dentro del algoritmo, los cuales fueron difíciles de descubrir porque no eran problemas como tal, sino que eran procesos mal enfocados en el algoritmo inicial.

### 5.1.3. Visualizadores

El uso de las visualizaciones de las mallas volumétricas usando GeomView fue una herramienta útil para poder entender como se iban comportando las mallas volumétricas con cada modificación que se realizaba en el algoritmo, esto ya que solo leyendo los datos entregados por el algoritmo Jens_Quality, se puede tener noción de la mejora de calidad obtenida, pero no con respecto a que la malla siga manteniendo la forma del objeto, por lo que era posible obtener mallas de buena calidad, pero que no representaban el modelo base usado.

### 5.1.4. Perspectiva y análisis

El entender como se estaba realizando el proceso iterativo y la forma de mover los nodos de la malla volumétrica, el cuál, era un enfoque individualista, es decir, que se intentaba

mejorar cada nodo individualmente intentando dejar a cada uno en su mejor condición antes de seguir con otro nodo, sirvió para entender por qué había que buscar nuevas formas de realizar el proceso de validación y mejoramiento de calidad.

La perspectiva lograda para poder mejorar la calidad de las mallas fue sin lugar a dudas el entender la malla como un sistema, donde cada uno de sus componentes y subcomponentes (elementos y nodos) dependen directamente de sus vecinos para la realización de cálculos, esto agregado al cambio del proceso iterativo, el cual se modificó para que cada nodo vaya mejorando un poco de forma secuencial, permitió mejorar la malla evitando que los nodos generaran problemas de ubicación entre si (refiriéndose al problema de que al mejorar la calidad de un nodo empeora la calidad de otro).

Descubrir el problema de validación cuando existen nodos negativos vecinos implicó que se realizase una excepción con estos nodos de la forma en que se repara, además, es posible que estos nodos no puedan ser reparados de la misma forma para diferentes mallas, por lo que lo ideal será realizar diferentes pruebas con diferentes formas de abordar estos subconjuntos de nodos. Además, se debe tener constancia de que la reparación de mallas no podrá solucionar todas las mallas geométricas por las complejidades que estas podrían tener.

### 5.1.5. Testeo

Para poder implementar los cambios en el algoritmo, se utilizó como prueba un hexaedro de ocho divisiones, esto ya que al ser una malla sencilla era fácil de modificar de forma manual para encontrar aquellas situaciones que generaran problemas.

La mejor opción para testear las implementaciones realizadas en el código fue realizar la automatización con el programa MeshGenerator para tener la cantidad de pruebas necesarias y ejecutar el jens_repair con cada una de estas pruebas. Además, es recomendable automatizar aún mas cierta información o cálculos que se requieran. Esto ya que actualmente, diferentes datos necesarios se encuentran en diferentes programas, lo cuál puede dificultar y enredar el testeo.

### 5.1.6. Resultado

Para la validación de las mallas, aquellas que no lograron ser resueltas, las causales fueron tanto porque se desplazan nodos negativos antes que el nodo causante de la invalidación o que la dirección invertida de desplazamiento no es hacia donde debiese ser desplazado. Por lo que para poder abordar esta problemática, se debiese buscar la manera de saber que nodos son los que hay que mover primero y buscar otra forma de poder desplazar al nodo que no sea la dirección calculada por el JENS o invertida a esta.

Para la mejora de calidad, calcular la calidad de los nodos incluyendo la de los vecinos, ade-

más de iterar sobre los nodos uno por uno para que estos se desplacen de a poco, permitió una mejora considerable en este proceso, llegando a valores superiores a 0.7 en calidad utilizando la métrica JENS.

## 5.2. Logros alcanzados

### 5.2.1. Objetivos Específicos

- Generar mallas geométricas con problemas que puedan estar en estado válido o inválido.

  La generación de mallas podía ser realizado usando un código que permitía recibir una malla superficial y generar el volumen de este. Sin embargo, el método que más se utilizó para poder hacer las pruebas y modificar con mayor precisión fue de forma manual, modificando directamente las coordenadas de los nodos a través del archivo de la malla. Además, gracias a la creación del programa MeshGenerator, se pudieron generar mallas con problemas de forma automática para poder hacer un testeo masivo.

- Medir la calidad de la superficie de la malla para verificar la validez y calidad de la malla geométrica.

  Al existir de antemano el código que permite medir la calidad de las mallas geométricas, utilizando la clase FaceContainer explicada en 2.7.2, se logró separar la superficie de la malla geométrica de su interior. Por lo que finalmente se logró medir la calidad de la superficie individualmente.

- Utilizar la métrica JENS en la superficie de la malla geométrica para desplazar los nodos defectuosos.

  El cálculo de la métrica JENS ya era realizada en el código jens_repair, y se realiza tanto para nodos internos como superficiales, sin embargo, la dirección calculada en un nodo con calidad negativa que esté conectado a otro nodo negativo permite mejora su calidad, sin embargo, esta no es la dirección hacia donde debiese ser desplazado el nodo, esta es la única situación donde se debió buscar otra forma de solucionarlo además del uso del JENS.

- Realizar el desplazamiento de nodos y vecinos de forma simultanea para secciones que no puedan ser resueltas moviendo solamente un nodo.

  Con respecto a este objetivo, como se mencionó en la sección 2.7.1 del marco conceptual, el código jens_repair ya permitía usar la métrica JENS para calcular la dirección de desplazamiento y así mover los nodos defectuosos, sin embargo, este fue modificado, llevándose la mayor parte del desarrollo de esta memoria, cambiando tanto el enfoque de la medición de calidad individual de los nodos a un enfoque de calidad del

sistema y el cambio iterativo de desplazamiento individual de cada nodo a uno consecutivo, de esta forma se logró mejorar la calidad de las mallas volumétricas de forma considerable. Esto permitió poder mover los nodos en conjunto en vez de individualmente como se hacía en sus inicios. Sin embargo, este proceso es ideal utilizarlo en mallas ya validadas.

- Medir la calidad de la malla geométrica final y verificar su validez.

### 5.2.2. Objetivo General

- El objetivo de esta memoria es implementar la métrica JENS para redistribuir los nodos de la superficie de una malla geométrica para así mejorar su calidad.

  Con los logros específicos alcanzados, y la solución implementada la cuál agrupa tanto los nodos internos como los superficiales, se puede concluir que el objetivo general fue logrado.

## 5.3. Trabajo futuro.

Como fue explayado en este documento, el principal problema que no se logró solucionar completamente en esta memoria fue la capacidad del Jens_repair de poder validar las mallas, esto quiere decir que todos sus elementos mejoren su calidad hasta que sean mayores que 0.

Como fue explicado, el cálculo de la dirección de desplazamiento realizado por el JENS para nodos con calidad negativa que estén conectados a otro nodo negativo con calidad negativa, a diferencia con los nodos de calidad positiva a los cuales la dirección calculada si es correcta, pueden no ser solucionados directamente, por lo que se propusieron métodos que puedan solucionarlos los cuales se deben mejorar.

Además, para quienes decidan continuar con esta investigación, será ideal poder entender el por qué la dirección calculada por el JENS no es la correcta para nodos negativos conectados a nodos vecinos también negativos.

La siguiente implementación importante, ideal para la velocidad de procesamiento del algoritmo es buscar la forma de detener las iteraciones realizadas por el algoritmo cuando no puede seguir mejorando la calidad de la malla. Esto ya que al cambiar la forma de iterar del algoritmo a través de los nodos, se volvió inválida la forma en que el algoritmo se detenía al no conseguir mejora, y actualmente el algoritmo realiza todas las iteraciones ingresadas como input, lo cuál puede causar problemas si la malla a mejorar tiene demasiados elementos.

Otra implementación no tan importante, pero que puede ser de utilizad es la siguiente.

Actualmente el código tiene la capacidad de generar un archivo .off del modelo que se ingresó, este formato puede ser leído por el programa GeomView, por lo que se podría implementar que cuando finalice la mejora de calidad del Jens_repair, la generación del archivo .off del modelo final para que muestre visualmente el resultado después de la mejora.

# ANEXOS

## 5.1. Explicación código implementación Jens_Repair

### 5.1.1. estructura

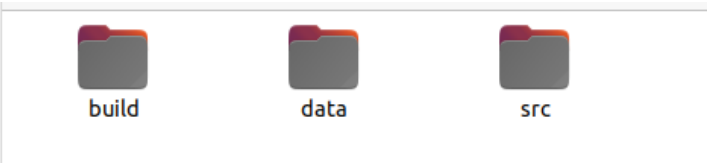


Figura 66: Estructura carpetas algoritmo Jens_Repair.
Fuente: Elaboración propia.

- build: Donde se genera el compilado del proyecto usando el makefile.
- data: Carpeta donde se guardan las mallas volumétricas a usar, tanto las .m3d como las .mdl.
- src: Contiene todo el código del programa.

### 5.1.2. Creación de archivo temporal .off para visualización

```
Main.cpp
}
//memorist

void headGeom(ofstream &output, int np, int nf){

    output << "OFF\n";
    output << np << " " << nf << " 0\n\n";
}

void dataGeom(ofstream &output, list<Point3D> &tpoints, list<Face> &tfaces){
    list<Point3D>::iterator iterp;
    list<Face>::iterator iterf;
    //print each point
    for(iterp=tpoints.begin();iterp!=tpoints.end();iterp++){
        Point3D p = *iterp;
        output << p[0] << " ";
        output << p[1] << " ";
        output << p[2] << "\n";
    }

    output << endl;

    //print each face
    for(iterf=tfaces.begin();iterf!=tfaces.end();iterf++){
        output << *iterf << " ";
        output << (*iterf).getColor() << endl;
    }
}
bool newTmpFile(string &stmp,ofstream &output){
    char tmp [16];
    int fd;

    strcpy(tmp,"geomviewXXXXXX\0");

    fd=mkstemp(tmp);

    if(fd==-1){
        cout << "  Error: Can't create a tmp file. Try saving";
        cout << " before viewing.\n";
        return false;
    }

    stmp+=tmp;
    output.open(tmp);
    return true;
}
```

Figura 67: Funciones para generar el archivo temportal .off parte 1.
Fuente: Elaboración propia.

```
Main.cpp
//save in tfaces only the faces that belong to one element from the given
//FaceContainer. Then it only save in tpoints the points used by the
//selected faces.
void clean(list<Point3D> &tpoints, vector<Point3D> &points, list<Face> &tfaces,
                    FaceContainer &tfc){

    vector<int> ptsidx (points.size(),-1);
    int usedpts = 0;
    for(int i=0;i<tfc.length();i++){
        if(tfc.getFace(i).numberOfElements()!=1)
            continue;
        vector<unsigned int> fpts = tfc.getFace(i).getPoints();
        for(int j=0;j<fpts.size();j++){
            if(ptsidx[fpts[j]]==-1){
                tpoints.push_back(points[fpts[j]]);
                ptsidx[fpts[j]]=usedpts;
                fpts[j]=usedpts++;
            }
            else{
                fpts[j]=ptsidx[fpts[j]];
            }
        }
        Face f(fpts);
        f.setColor(tfc.getFace(i).getColor());
        tfaces.push_back(f);
    }
}

//-----------------------------------------------------------------
//-----------------------------------------------------------------

void cleanUnusedNodes(list<Point3D> &tpoints,vector<Point3D> &points,
                                list<Face> &tfaces,
                                FaceContainer &tfc){

    vector<int> ptsidx (points.size(),-1);
    int usedpts = 0;
    for(int i=0;i<tfc.length();i++){
        vector<unsigned int> fpts = tfc.getFace(i).getPoints();
        for(int j=0;j<fpts.size();j++){
            if(ptsidx[fpts[j]]==-1){
                tpoints.push_back(points[fpts[j]]);
                ptsidx[fpts[j]]=usedpts;
                fpts[j]=usedpts++;
            }
            else{
                fpts[j]=ptsidx[fpts[j]];
            }
        }
        Face f(fpts);
        f.setColor(tfc.getFace(i).getColor());
        tfaces.push_back(f);
    }
}
```

Figura 68: Funciones para generar el archivo temportal .off parte 2.
Fuente: Elaboración propia.

La ejecución de estas funciones esta comentada en la función main() dentro del archivo

main.cpp al no haber sido usadas en lo implementado.

### 5.1.3. Separar las caras superficiales de la malla volumétrica

```
470
471     FaceContainer surface_fc;
472     //imprimimos las caras de la superficie
473     int cont=0;
474     bool element=false;
475
476     for (unsigned int i=0; i<elements.size(); i++){
477         //if (elements[i]->surfaceElement==true)
478             elements[i]->ESChecker(all_faces, surface_fc);
479         }
480
481     cout<<"this is the total of faces of surfaces elements"<<endl;
482     cout<<cont<<endl;
```

Figura 69: Ejecución de la función para separar las caras superficiales de la malla volumétrica.
Fuente: Elaboración propia.

Esta ejecución se realiza en la función Main() dentro del archivo main.cpp

```
113 void Element::ESChecker(FaceContainer &fc, FaceContainer &sfc) {
114     list<unsigned int>::iterator iter;
115     list<unsigned int>::iterator iter2;
116     for(iter=faces.begin();iter!=faces.end();iter++){
117         if(fc.getFace(*iter).numberOfElements()==1){
118             surfaceElement=true;
119             //Element=true;
120             for(iter2=faces.begin();iter2!=faces.end();iter2++){
121                 sfc.addFace(fc.getFace(*iter2));
122                 }
123             break;
124             //fc.getFace(*iter).setColor(color);
125             //sfc.(fc.getFace(*iter));
126
127         }
128     }
129
130
131
132 }
133 //--
134 //-----------------------------------------------------------------//
135 //-----------------------------------------------------------------//
```

Figura 70: Función para saber si una cara es superficial.
Fuente: Elaboración propia.

Esta función está definida en el archivo Element.cpp

```
519
520     //projection
521     for (auto node: inv_nodesS){
522         Clobscode::Point3D p = points[node];
523         points[node] = Clobscode::TriMesh(tri_pts, tri_allfaces).getProjection(p);
524     }
525
```

Figura 71: Ejecución de la función projection() en los nodos inválidos superficiales.
Fuente: Elaboración propia.
Realizado en la función Main() dentro del archivo main.cpp

### 5.1.4. Obtención de nodos superficiales

```
//Esta función agrega los nodos pertenecientes a la superficie a una lista especial para estos y tambien mide el step necesario para estos.
void Quality::InvPointSFChecker2(vector<Point3D> &p, vector<Element *> &e, list<unsigned int> &inv_nodesS, list<unsigned int> &inv_nodes_mid,
 vector<tuple<int,int>> &inv_nodesS_steps, vector<tuple<int,int>> &inv_nodes_mid_steps, list<unsigned int> &inv_nodes,
 vector<tuple<int,int, string>> &inv_nodes_steps, FaceContainer &fc,   float &step, unsigned  int &i,  unsigned  int &j ,
 bool belong, map<int, list<int>> &neighborsmap){

    list<unsigned int>::iterator iter;

    //recorremos las caras del elemento recorrido y usamos el face container para verificar si son caras superficiales.
    //Verificamos si estamos dentro de un elemento superficial.
    bool flag=false;
    for(iter=e[i]->faces.begin();iter!=e[i]->faces.end();iter++){

            if(fc.getFace(*iter).numberOfElements()==1){
                flag=true;
                break;
            }


        }
        if (flag){
            for(iter=e[i]->faces.begin();iter!=e[i]->faces.end();iter++){
            //tomamos la cara superficial
            if(fc.getFace(*iter).numberOfElements()==1){
                vector<unsigned int> f_points = fc.getFace(*iter).getPoints();
                //recorremos los nodos de la cara
                for (unsigned int fp=0; fp<f_points.size(); fp++){
                    //agregamos el nodo si lo encontramos.
                    if (e[i]->getPoints()[j] == f_points[fp]){
                        //agregamos el nodo a la lista superficial y a la lista total
                        inv_nodesS.push_back(e[i]->getPoints()[j]);
                        inv_nodes.push_back(e[i]->getPoints()[j]);

                        //calculamos el step, agregandolo a una lista de tuplas (indexNode, step) solo si no han sido agregados.
                        Point3D p1 = p[e[i]->getPoints()[j]];

                        //cout<<"largo de listas de vecinos de: "<<e[i]->getPoints()[j]<<"es: "<<neighborsmap.find(e[i]->getPoints()[j])->sec
                        for (auto node: neighborsmap.find(e[i]->getPoints()[j])->second){


                            Point3D p2 = p[node];
                            step+=p1.distance(p2);

                        }


                        step /=(neighborsmap[e[i]->getPoints()[j]]).size();

                        /*for(unsigned int point=0; point!=e[i]->getPoints().size();point++ ){
                            Point3D p2 = p[e[i]->getPoints()[point]];
                            if (p1.distance(p2)<step && j!=point)
                                step=p1.distance(p2);
                                //cout<<"---step: "<<step<<endl;
```

Figura 72: Función obtener los nodos superficiales parte 1.
Fuente: Elaboración propia.

```
                        for (const auto &node: inv_nodes_steps){
                            if (e[i]->getPoints()[j]==get<0>(node))
                                belong=true;

                        }

                        if (!belong){
                            cout<<"es superficial"<<step<<endl;
                            inv_nodes_steps.push_back(tuple<int,int, string>(e[i]->getPoints()[j], step, "sup"));
                            cout<<"index: "<<e[i]->getPoints()[j]<<"distance: "<<step<<endl;
                        }


                        break;
                    }
                }
            } else {

                vector<unsigned int> f_points = fc.getFace(*iter).getPoints();
                //recorremos los nodos de la cara
                for (unsigned int fp=0; fp<f_points.size(); fp++){
                    //agregamos el nodo si lo encontramos.
                    if (e[i]->getPoints()[j] == f_points[fp]){
                        //cout<<"encontre el nodo en la superficie y se agrega a inv_nodesS"<<endl;
                        inv_nodes_mid.push_back(e[i]->getPoints()[j]);
                        inv_nodes.push_back(e[i]->getPoints()[j]);
                        //calculamos el step, agregandolo a una lista de tuplas (indexNode, step) solo si no han sido agregados.
                        Point3D p1 = p[e[i]->getPoints()[j]];

                        for (auto node: neighborsmap.find(6)->second){

                            Point3D p2 = p[node];
                            step+=p1.distance(p2);

                        }


                        step /=(neighborsmap[e[i]->getPoints()[j]]).size();

                        /*for(unsigned int point=0; point!=e[i]->getPoints().size();point++ ){
                            Point3D p2 = p[e[i]->getPoints()[point]];
                            if (p1.distance(p2)<step && j!=point)
                                step=p1.distance(p2);
                                //cout<<"---step: "<<step<<endl;
                        }*/

                        for (const auto &node: inv_nodes_steps){
                            if (e[i]->getPoints()[j]==get<0>(node))
                                belong=true;

                        }
```

Figura 73: Función obtener los nodos superficiales parte 2.
Fuente: Elaboración propia.

```
                    if (!belong){
                        cout<<"es mid"<<step<<endl;
                        inv_nodes_steps.push_back(tuple<int,int, string>(e[i]->getPoints()[j],step, "mid"));
                    }


                    break;
                }
            }
        }

        }


}
}
```

Figura 74: Función obtener los nodos superficiales parte 3.
Fuente: Elaboración propia.

### 5.1.5. Cambio proceso iterativo

En la función Quality_repair definida en el archivo Quality.cpp, es donde se realizó el cambio del proceso iterativo para desplazar los nodos.

Esta función tiene implementada tanto la validación como la mejora de calidad de la malla. La validación quedó comentada por el problema encontrado en la sección Validación de la solución.

```
Main.cpp    Quality.cpp    Element.cpp

bool Quality::repair(vector<Point3D> &p, list<unsigned int> &inv_nodesS, vector<Element *> &e,
                     Metric *metric, unsigned int &max_iter, const double &threshold, FaceContainer &fc, map<int, list<int>> &neighborsmap){



    //This method can be called several times with different values for
    //threshold. Only the fisrt time we want to link elements to nodes.
    if (nodeElements.empty()) {
        linkElementsToNodes(e);
    }


    //Make a list with all unacceptable nodes
    list<unsigned int> inv_nodes;
    vector<tuple<int,int, string>> inv_nodes_steps;
    list<unsigned int> inv_nodes_mid;
    vector<tuple<int,int>> inv_nodes_mid_steps;
    vector<tuple<int,int>> inv_nodesS_steps;
    float step= 0;
    for (unsigned int i=0;i<e.size();i++){

        vector<double> node_jens = metric->getQuality(e[i],p);
        for (unsigned int j=0;j<node_jens.size();j++) {
            //
            if (node_jens[j] < threshold) {
                cout<<"threshold: "<<threshold<<endl;
                cout<<"este es el valor de este nodo: "<<node_jens[j]<<endl;
                step= 0;
                //aca va la nueva función
                bool belong=false;
                InvPointSFChecker2(p, e, inv_nodesS, inv_nodes_mid, inv_nodesS_steps, inv_nodes_mid_steps, inv_nodes, inv_nodes_steps, fc, st
                }



        }
    }
    cout<<"termino los primeros nodos"<<endl;
    inv_nodesS.sort();
    inv_nodesS.unique();

    inv_nodes_mid.sort();
    inv_nodes_mid.unique();

    for (unsigned int i=0;i<e.size();i++){

        //vector<double> node_jens = e[i]->getJENS(p);
        vector<double> node_jens = metric->getQuality(e[i],p);
        for (unsigned int j=0;j<node_jens.size();j++) {
            //
            if (node_jens[j] < threshold) {
                step=0;
                bool belong=false;
```

Figura 75: Función para desplazar los nodos a mejorar parte 1.
Fuente: Elaboración propia.

```
Main.cpp    Quality.cpp    Element.cpp

                bool inter = true;

                for (auto node: inv_nodes){
                    if (e[i]->getPoints()[j]==node )
                        inter=false;
                }

                if (inter==true){
                    inv_nodes.push_back(e[i]->getPoints()[j]);
                    Point3D p1 = p[e[i]->getPoints()[j]];

                    for (auto node: neighborsmap.find(e[i]->getPoints()[j])->second){

                            Point3D p2 = p[node];
                            step+=p1.distance(p2);

                        }

                        step /=(neighborsmap[e[i]->getPoints()[j]]).size();

                        for (const auto &node: inv_nodes_steps){
                            if (e[i]->getPoints()[j]==get<0>(node))
                                belong=true;

                        }

                        if (!belong){
                            cout<<"es interno"<<step<<endl;
                            inv_nodes_steps.push_back(tuple<int,int, string>(e[i]->getPoints()[j],step, "inter"));
                            cout<<"index: "<<e[i]->getPoints()[j]<<"distance: "<<step<<endl;
                        }
                }

                }

        }
    }
    inv_nodes.sort();
    inv_nodes.unique();

    cout<<"termina todos los sorts"<<endl;



    //Repair them.
    bool valid = true;


    string type;
    int potenciador;

    /*
    for (int i=0; i<max_iter; i++){
        valid=true;
```

Figura 76: Función para desplazar los nodos a mejorar parte 2.
Fuente: Elaboración propia.

```
Main.cpp    Quality.cpp    Element.cpp

        for (auto node: inv_nodes){
            for (auto tuple: inv_nodes_steps){
                if (node == get<0>(tuple)){
                    step=get<1>(tuple);
                    type=get<2>(tuple);

                }

                }
            if (!repairOne(p,e,node,metric,max_iter,0, step, "valid", neighborsmap)) {
                valid = false;
            }

        }

    }
    */




    cout << " >Number of unacceptable nodes: "<< inv_nodes.size() << endl;
    //cout << " >Number of unacceptable superficial nodes: "<< inv_nodesS.size() << endl;
    if (!valid) {
        cout << "  Some nodes could not be repaired to achieve the threshold.\n";
        cout << "  The process finished. Resulting mesh written to output.\n";
        return false;
    }
    else {
        cout << "  All elements achieved the threshold.\n";
    }


    //Detect elements with node quality below the threshold
    bool quality = true;
```

Figura 77: Función para desplazar los nodos a mejorar parte 3.
Fuente: Elaboración propia.

```
Main.cpp    Quality.cpp    Element.cpp
347     unsigned int acceptable = 0;
348
349     //descomentar cuando esté listo.
350
351     for (int i=0; i<max_iter; i++){
352
353         for (unsigned int i=0;i<e.size();i++){
354             vector<double> node_jens = metric->getQuality(e[i],p);
355             double worst = 1;
356             unsigned int widx = 0, last_time = UINT_MAX;
357             //find the worst node within this element
358             for (unsigned int j=0;j<node_jens.size();j++) {
359                 double current = node_jens[j];
360                 if (current < threshold && current < worst) {
361                     worst = current;
362                     widx = j;
363                 }
364             }
365             //while (worst<threshold) {
366                 if (widx==last_time) {
367                     break;
368                 }
369                 if (worst < threshold) {
370                     acceptable++;
371                     for (auto tuple: inv_nodes_steps){
372                         if (e[i]->getPoints()[widx] == get<0>(tuple)){
373                             //cout<<"nodo encontrado: "<<get<0>(tuple)<<endl;
374
375                             step=get<1>(tuple);
376                             type=get<2>(tuple);
377                         }
378
379                     }
380
381                     if (!repairOne(p,e,e[i]->getPoints()[widx],metric,max_iter,threshold, step*0.3, "quality", neighborsmap)) {
382                         quality = false;
383                     }
384                     last_time = widx;
385                 }
386                 worst = 2;
387                 node_jens=metric->getQuality(e[i],p);
388                 for (unsigned int j=0;j<node_jens.size();j++) {
389                     double current = node_jens[j];
390                     if (current < worst) {
391                         worst = current;
392                         widx = j;
393                     }
394                 }
395
396
397             //}
398         }
399     }
400     cout << " >Number of nodes below target threshold: "<< acceptable << endl;
```

Figura 78: Función para desplazar los nodos a mejorar parte 4.
Fuente: Elaboración propia.

```
401     if (!quality) {
402         cout << "  Some nodes did not reach target threshold\n";
403         cout << "  due to the number of max iterations was too small.\n";
404         cout << "  The process finished. Resulting mesh written to output\n";
405         return false;
406     }
407     else {
408         cout << "  All elements reached target quality.\n";
409     }
410
411
412     return true;
413 }
414
```

Figura 79: Función para desplazar los nodos a mejorar parte 5.
Fuente: Elaboración propia.

### 5.1.6. Mejorar la calidad del nodo a desplazar y sus vecinos

Repair_One, la cual es la encargada de aplicar la métrica JENS es donde se realiza el desplazamiento de los nodos inválidos, en esta función se realizó el cambio nombrado en la sección 3.2.7

```
Main.cpp    Quality.cpp    Element.cpp

416  bool Quality::repairOne(vector<Point3D> &p, vector<Element *> &e,
417                          unsigned int &nodeidx, Metric *metric,
418                          unsigned int &max_iter, const double &threshold, int step, string type, map<int, list<int>> &neighborsmap) {
419
420
421      //Compute current quality considering all elements attached to this node
422      //that don't achive the threshold.
423
424      cout <<"node: "<<nodeidx<<endl;
425      int totalneighbors=0;
426      double quality_sum=0;
427      double current_qua = 0;
428      unsigned int num_inv = 0;
429      for (auto node: neighborsmap.find(nodeidx)->second){
430          current_qua = 0;
431          num_inv = 0;
432          list<tuple<unsigned int,unsigned int> > elemlist = nodeElements[node];
433
434          for (auto elem: elemlist) {
435              //compute the quality within this element
436              //double qua = e[std::get<0>(elem)]->getJENS(p)[std::get<1>(elem)];
437              double qua = metric->getQuality(e[std::get<0>(elem)],p)[std::get<1>(elem)];
438              if (qua<threshold) {
439                  //cout<<"este elemento interno tiene calidad bája"<<endl;
440                  current_qua += qua;
441                  num_inv++;
442                  //cout << "num_inv: "<< num_inv << " (" << std::get<0>(elem) << ",";
443                  //cout << std::get<1>(elem) << ")\n";
444              }
445
446          }
447          if (num_inv>0) {
448              current_qua /= num_inv;
449              quality_sum+= current_qua;
450              totalneighbors++;
451          }
452
453
454
455      }
456
457
458
459
460      if (totalneighbors>0) {
461          quality_sum/= totalneighbors;
462          current_qua=quality_sum;
463      }
464      else {
465
466          return true;
467      }
468
469
```

Figura 80: Función para desplazar los nodos a mejorar parte 1.
Fuente: Elaboración propia.

```
Main.cpp    Quality.cpp    Element.cpp

470
471
472      //compute gradient for this node:
473      //step = 1
474      double epsilon = 0.001 , new_qua = 0;
475      unsigned int iter = 0;
476
477      cout << "   >for node " << nodeidx;
478      cout << " initial pos " << p[nodeidx] <<"current neighborgs quality"<< " (" << current_qua << ")\n";
479
480
481          num_inv = 0;
482
483          //compute delta in X:
484          p[nodeidx][0] += epsilon;
485          new_qua = 0;
486          quality_sum=0;
487          totalneighbors=0;
488  ////////////////////////////////
489          for (auto node: neighborsmap.find(nodeidx)->second){
490              new_qua = 0;
491              num_inv = 0;
492              list<tuple<unsigned int,unsigned int> > elemlist = nodeElements[node];
493
494              for (auto elem: elemlist) {
495                  //compute the quality within this element
496                  //double qua = e[std::get<0>(elem)]->getJENS(p)[std::get<1>(elem)];
497                  double qua = metric->getQuality(e[std::get<0>(elem)],p)[std::get<1>(elem)];
498                  if (qua<threshold) {
499                      //cout<<"este elemento interno tiene calidad bája"<<endl;
500                      new_qua += qua;
501                      num_inv++;
502                      //cout << "num_inv: "<< num_inv << " (" << std::get<0>(elem) << ",";
503                      //cout << std::get<1>(elem) << ")\n";
504                  }
505
506              }
507              if (num_inv>0) {
508                  new_qua /= num_inv;
509                  quality_sum+= new_qua;
510                  totalneighbors++;
511              }
512
513
514
515          }
516
517          if (totalneighbors>0) {
518              quality_sum/= totalneighbors;
519              new_qua=quality_sum;
520          }
521
522
523          cout<< "new_qua: "<<new_qua<<endl;
```

Figura 81: Función para desplazar los nodos a mejorar parte 2.
Fuente: Elaboración propia.

```
Main.cpp    Quality.cpp    Element.cpp
524         cout<< "current qua: "<<current_qua<<endl;
525
526         double deltaX = (new_qua - current_qua)/epsilon;
527         p[nodeidx][0] -= epsilon;
528
529         //compute delta in Y:
530         p[nodeidx][1] += epsilon;
531         new_qua = 0;
532
533         quality_sum=0;
534         totalneighbors=0;
535 ////////////////////////////////
536         for (auto node: neighborsmap.find(nodeidx)->second){
537             new_qua = 0;
538             num_inv = 0;
539             list<tuple<unsigned int,unsigned int> > elemlist = nodeElements[node];
540
541             for (auto elem: elemlist) {
542                 //compute the quality within this element
543                 //double qua = e[std::get<0>(elem)]->getJENS(p)[std::get<1>(elem)];
544                 double qua = metric->getQuality(e[std::get<0>(elem)],p)[std::get<1>(elem)];
545                 if (qua<threshold) {
546                     //cout<<"este elemento interno tiene calidad bája"<<endl;
547                     new_qua += qua;
548                     num_inv++;
549                     //cout << "num_inv: "<< num_inv << " (" << std::get<0>(elem) << ",";
550                     //cout << std::get<1>(elem) << ")\n";
551                 }
552
553             }
554             if (num_inv>0) {
555                 new_qua /= num_inv;
556                 quality_sum+= new_qua;
557                 totalneighbors++;
558             }
559
560
561
562         }
563
564         if (totalneighbors>0) {
565             quality_sum/= totalneighbors;
566             new_qua=quality_sum;
567         }
568
569
570
571         double deltaY = (new_qua - current_qua)/epsilon;
572         p[nodeidx][1] -= epsilon;
573
574         //compute delta in Z:
575         p[nodeidx][2] += epsilon;
576         new_qua = 0;
577         quality_sum=0;
```

Figura 82: Función para desplazar los nodos a mejorar parte 3.
Fuente: Elaboración propia.

```
Main.cpp    Quality.cpp    Element.cpp
578         totalneighbors=0;
579 ////////////////////////////////
580         for (auto node: neighborsmap.find(nodeidx)->second){
581             new_qua = 0;
582             num_inv = 0;
583             list<tuple<unsigned int,unsigned int> > elemlist = nodeElements[node];
584
585             for (auto elem: elemlist) {
586                 //compute the quality within this element
587                 //double qua = e[std::get<0>(elem)]->getJENS(p)[std::get<1>(elem)];
588                 double qua = metric->getQuality(e[std::get<0>(elem)],p)[std::get<1>(elem)];
589                 if (qua<threshold) {
590                     //cout<<"este elemento interno tiene calidad bája"<<endl;
591                     new_qua += qua;
592                     num_inv++;
593                     //cout << "num_inv: "<< num_inv << " (" << std::get<0>(elem) << ",";
594                     //cout << std::get<1>(elem) << ")\n";
595                 }
596
597             }
598             if (num_inv>0) {
599                 new_qua /= num_inv;
600                 quality_sum+= new_qua;
601                 totalneighbors++;
602             }
603
604
605
606         }
607
608         if (totalneighbors>0) {
609             quality_sum/= totalneighbors;
610             new_qua=quality_sum;
611         }
612
613
614
615         double deltaZ = (new_qua - current_qua)/epsilon;
616         p[nodeidx][2] -= epsilon;
617
618         Point3D direction(deltaX,deltaY,deltaZ);
619         direction.normalize();
620         cout<<"la dirección de desplazamiento del nodo: "<<nodeidx<< " es: "<<direction<<endl;
621         Point3D d = direction*step;
622         int iterator=0;
623         double auxQua=-1;
624         Point3D nodeAux=p[nodeidx];
625         int iteratorSaver=0;
626         double current_qua2 = 0;
627
628
629         //step = step-step*(1/max_iter);
630         current_qua2=0;
631         num_inv = 0;
```

Figura 83: Función para desplazar los nodos a mejorar parte 4.
Fuente: Elaboración propia.

```
Main.cpp    Quality.cpp    Element.cpp

632         double one_qua = 0;
633         p[nodeidx] += direction*step;
634
635
636
637 ////////////////////////////////
638         totalneighbors=0;
639         quality_sum=0;
640
641 for (auto node: neighborsmap.find(nodeidx)->second){
642         current_qua = 0;
643         num_inv = 0;
644         list<tuple<unsigned int,unsigned int> > elemlist = nodeElements[node];
645
646         for (auto elem: elemlist) {
647             //compute the quality within this element
648             //double qua = e[std::get<0>(elem)]->getJENS(p)[std::get<1>(elem)];
649             double qua = metric->getQuality(e[std::get<0>(elem)],p)[std::get<1>(elem)];
650             if (qua<threshold) {
651                 //cout<<"este elemento interno tiene calidad bája"<<endl;
652                 current_qua += qua;
653                 num_inv++;
654                 //cout << "num inv: "<< num_inv << " (" << std::get<0>(elem) << ",";
655                 //cout << std::get<1>(elem) << ")\n";
656             }
657             else {
658                 one_qua = qua;
659             }
660
661         }
662         if (num_inv>0) {
663             current_qua /= num_inv;
664             quality_sum+= current_qua;
665             totalneighbors++;
666         }
667
668
669
670     }
671     if (totalneighbors>0) {
672             quality_sum/= totalneighbors;
673             current_qua=quality_sum;
674         }
675
676
677
678
679
680
681 /////////////////
682
683
684
685
```

Figura 84: Función para desplazar los nodos a mejorar parte 5.
Fuente: Elaboración propia.

```
682
683
684
685
686
687             iter++;
688             iterator++;
689
690
691
692     current_qua2=current_qua;
693
694
695     cout << "    final pos " << p[nodeidx] << endl;
696
697     cout << "    for node " << nodeidx << " final quality is " << current_qua2;
698     cout << " in " << iter << " iterations\n";
699
700
701
702
703     if (current_qua2<threshold) {
704         cout << "Warning: for node " << nodeidx << " max iteration reached without";
705         cout << " achieving targer threshold (" << threshold << ")\n";
706         return false;
707     }
708
709     return true;
710 }
711
712
713
714
715 //-----------------------------------------------------------------------
716 //-----------------------------------------------------------------------
```

Figura 85: Función para desplazar los nodos a mejorar parte 6.
Fuente: Elaboración propia.

# REFERENCIAS BIBLIOGRÁFICAS

[Bucki *et al.*, 2010] Bucki, M., Lobos, C., y Payan, Y. (2010). A fast and robust patient specific finite element mesh registration technique: application to 60 clinical cases. *Medical image analysis*, 14(3):303–317.

[Bucki *et al.*, 2011] Bucki, M., Lobos, C., Payan, Y., y Hitschfeld, N. (2011). Jacobian-based repair method for finite element meshes after registration. *Engineering with Computers*, 27(3):285–297.

[C. *et al.*, 2022] C., L., C., A., E., D., y N., H. (2022). Measuring geometrical quality of different 3d linear element types. *Numerical Algorithms*, 90(1):423–446.

[Kwok y Chen, 2000] Kwok, W. y Chen, Z. (2000). A simple and effective mesh quality metric for hexahedral and wedge elements. En *IMR*, pp. 325–333.

[Lobos y González, 2015] Lobos, C. y González, E. (2015). Mixed-element octree: a meshing technique toward fast and real-time simulations in biomedical applications. *International journal for numerical methods in biomedical engineering*, 31(12).

[Zienkiewicz *et al.*, 2013] Zienkiewicz, O. C., Taylor, R. L., y Zhu, J. Z. (2013). *The finite element method: its basis and fundamentals*. Elsevier.