Matias Nicolas Benary

Desarrollo de una base de datos inspirada en imágenes médicas para uso en técnicas de Imágenes por Microondas

2024

Instituto de Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Informe Final

Práctica Profesional Supervisada

Universidad Nacional Arturo Jauretche
Carrera de Ingeniería en Informática

Matias Nicolas Benary

Florencio Varela, Diciembre 2024

Desarrollo de una base de datos inspirada en imágenes médicas para uso en técnicas de Imágenes por Microondas.

ESTUDIANTE

Apellido y Nombres: Matias Nicolas Benary
Correo electrónico: matiasbenary@gmail.com

ORGANIZACIÓN DONDE SE REALIZA LA PRÁCTICA PROFESIONAL SUPERVISADA

Nombre de la institución: *Universidad Nacional Arturo Jauretche*
Dirección: *Av. Calchaquí 6200, Florencio Varela, (1888) Buenos Aires, Argentina*
Teléfono: *+54 11 4275-6100*
Sector: *Programa Tecnologías de la Información y la Comunicación (TIC) en aplicaciones de interés social, Instituto de Ingeniería y Agronomía*

TUTOR DE LA ORGANIZACIONAL

Apellido y Nombres: *Prof. Mg. OSIO, Jorge*
Correo electrónico: josio@unaj.edu.ar

DOCENTE SUPERVISOR

Apellido y Nombres: *Dr. Irastorza Ramiro Miguel*
Correo electrónico: ramiro.m.irastorza@gmail.com

COORDINADOR DE LA CARRERA DE INGENIERÍA INFORMÁTICA

Apellido y Nombres: *Dr. Ing. Morales, Martín*
Correo electrónico: martin.morales@unaj.edu.ar

1. Agradecimientos

Quiero expresar mi más profundo agradecimiento a todas las personas que, de una u otra manera, han sido parte fundamental de este camino lleno de aprendizajes y logros.

En primer lugar, a mi pareja, quien ha sido mi soporte incondicional durante todo este proceso. Su amor, paciencia y apoyo constante me han dado la fuerza para superar los obstáculos y continuar adelante en momentos difíciles.

A mis padres, quienes siempre creyeron en mí y me han brindado su apoyo inquebrantable. Sus valores y enseñanzas han sido la base de mi crecimiento personal y profesional.

A mi universidad, por ser el espacio donde pude desarrollar mis conocimientos y habilidades, y por su compromiso con la educación y la igualdad de oportunidades. Gracias a la UNAJ por brindarme la oportunidad de formarme en un ambiente de excelencia.

A Martín Morales, por brindarme la oportunidad de realizar este proyecto final, y a Ramiro Irastorza, mi tutor, por su guía experta, paciencia y apoyo incondicional a lo largo de todo el proceso. Su orientación ha sido esencial para el desarrollo de este trabajo. Gracias por ser un verdadero ejemplo a seguir, demostrando que, a pesar de los desafíos, siempre se debe trabajar con determinación por los sueños y el amor al arte.

A todos los profesores que me acompañaron durante mi formación, quienes compartieron su sabiduría y pasión por la enseñanza. Gracias por inspirarme a seguir aprendiendo y por los desafíos que me ayudaron a superar.

A mis compañeros y amigos de la facultad, quienes fueron parte esencial de este viaje. A todos con quienes compartí horas de estudio, risas y desafíos. Su amistad y colaboración han sido una fuente constante de motivación y aprendizaje. Es un pilar muy necesario para llegar lejos.

A mis jefes y compañeros de trabajo, por su colaboración, apoyo y por brindarme un entorno enriquecedor donde pude aplicar y expandir mis conocimientos. Gracias por creer en mí y por su apoyo constante.

Finalmente, a todos mis seres queridos, cuyo amor, aliento y apoyo, me han permitido llegar hasta aquí. Este logro es también el resultado de su contribución incondicional.

2. Resumen

La presente Práctica Profesional Supervisada (PPS) se desarrollará en el marco del proyecto de la Universidad Nacional Arturo Jauretche (UNAJ), titulado “Desarrollo de una base de datos inspirada en imágenes médicas para uso en técnicas de Imágenes por Microondas (TMO)”, bajo la supervisión del Ing. Dr. Ramiro Miguel Irastorza.

El objetivo del proyecto es el desarrollo de una base de datos basada en imágenes médicas, orientada a su aplicación en técnicas de Imágenes por Microondas. El enfoque principal radica en la creación de modelos CAD semiautomáticos a partir de imágenes DICOM, mediante la utilización de algoritmos como Marching Squares para la segmentación de tejidos, tales como hueso cortical, hueso trabecular y piel.

La herramienta desarrollada en Python permite procesar dichas imágenes y generar archivos .geo compatibles con el software Gmsh, con el fin de realizar simulaciones numéricas basadas en el Método de Elementos Finitos (FEM).

Palabras Clave: TMO,FEM,DICOM,FDTD,CAD,Marching Squares

2.1. Abstract

This Supervised Professional Practice (PPS) will be carried out within the framework of the project at the National University Arturo Jauretche (UNAJ), titled "Development of a Database Inspired by Medical Images for Use in Microwave Imaging Techniques (TMI)", under the supervision of Eng. Dr. Ramiro Miguel Irastorza.

The objective of the project is to develop a database based on medical images, aimed at applications in Microwave Imaging Techniques. The main focus lies in the creation of semi-automatic CAD models from DICOM images, using algorithms such as Marching Squares to segment tissues including cortical bone, trabecular bone, and skin.

The tool developed in Python processes these images and generates .geo files compatible with Gmsh software, enabling numerical simulations based on the Finite Element Method (FEM).

Keywords: TMO,FEM,DICOM,FDTD,CAD,Marching Squares

Índice

3. Introducción

Esta Práctica Profesional Supervisada (PPS) se desarrolla en el marco del proyecto UNAJ Investiga 2023: "Algoritmos de machine learning para el procesamiento de imágenes en aplicaciones biomédicas, agronómicas y ambientales". Dentro de la línea de investigación en aplicaciones biomédicas, el enfoque se centra en el estudio de métodos alternativos para la evaluación no invasiva de la salud ósea, específicamente mediante el uso de Tomografía de Microondas. Para la reconstrucción de las imágenes, se emplean algoritmos supervisados de machine learning, lo que requiere la creación de bases de datos. En este contexto, dichas bases de datos se generan mediante simulaciones de modelos basados en geometrías reales. A continuación, se describen los aspectos fundamentales de este problema junto con los objetivos específicos establecidos para el desarrollo de esta PPS.

3.1. Presentación del problema y su relevancia.

La salud ósea es un aspecto crucial de la calidad de vida humana, y las enfermedades que afectan la densidad y la integridad de los huesos, como la osteoporosis, representan un desafío médico significativo. La osteoporosis, en particular, es una afección silenciosa y generalmente asintomática en sus primeras etapas, lo que la convierte en un gran enemigo de la salud humana [1]. Esta enfermedad se caracteriza por la pérdida gradual de la densidad ósea y el deterioro de la microestructura del tejido óseo, lo que aumenta la fragilidad de los huesos y el riesgo de fracturas.

Es fundamental detectar signos tempranos de la osteoporosis y monitorear su progresión para intervenir de manera preventiva y mejorar la calidad de vida de los pacientes. Contar con herramientas de diagnóstico precisas y no invasivas es esencial en este contexto.

La elección de la tomografía por microondas (TMO) [2, 3] , está impulsada por la necesidad de una evaluación cuantitativa de la densidad ósea sin recurrir a radiaciones ionizantes. Este enfoque ofrece una ventaja significativa en términos de seguridad para el paciente. La TMO ha emergido como una técnica prometedora en el campo de la bioingeniería y la medicina debido a su capacidad para obtener imágenes no invasivas y de bajo costo, lo que la hace especialmente atractiva en comparación con otras técnicas de diagnóstico médico, como la resonancia magnética. Además, la adaptabilidad de la técnica a diversas configuraciones la hacen apta para equipamiento ambulatorio (por ejemplo en otras aplicaciones como la evaluación de accidentes cerebro vasculares [4]).

En aplicaciones de salud ósea, la tomografía por microondas se posiciona

como un complemento valioso a métodos más precisos, como la absorciometría de rayos X de energía dual (DXA o DEXA), también conocida como densitometría ósea, fortaleciendo así la capacidad de diagnóstico y tratamiento de enfermedades óseas. Esta tecnología utiliza microondas para iluminar objetos biológicos, aprovechando las diferencias en sus propiedades dieléctricas[1] para generar imágenes a partir de los campos dispersados por este contraste.

3.2. Breve explicación de por qué se eligió la tomografía por microondas

La elección de la tomografía por microondas se basa en una evaluación exhaustiva de las distintas técnicas de imagenología médica disponibles. Para comprender plenamente esta elección, es esencial abordar el concepto de radiación ionizante y no ionizante. La radiación ionizante, presente en técnicas como la tomografía computarizada (TC) y la radiografía convencional, posee la energía suficiente para ionizar átomos y moléculas, lo que puede alterar estructuras celulares y causar daño biológico. Aunque estas técnicas ofrecen una resolución excepcional, su uso conlleva riesgos asociados a la exposición prolongada a esta forma de radiación. En contraste, la radiación no ionizante, a la cual pertenecen las microondas, no tiene la energía necesaria (por su menor frecuencia) para alterar la estructura atómica y molecular, lo que la hace inherentemente más segura para su aplicación en el ámbito médico. La tomografía por microondas se basa en el uso de este tipo de radiación, lo que elimina los riesgos asociados con la exposición a radiación ionizante, convirtiéndola en una opción más segura y sostenible. Es importante destacar que el espectro electromagnético abarca una amplia gama de ondas, desde las de mayor energía y frecuencia, como los rayos X y los rayos gamma, hasta las de menor energía y frecuencia, como las microondas y las ondas de radio (ver Fig. 1). Cada tipo de radiación tiene características únicas que determinan su interacción con la materia y su idoneidad para aplicaciones específicas en medicina y otras disciplinas.

La tomografía por microondas se distingue por su capacidad para proporcionar información sobre las propiedades dieléctricas de los tejidos. Esta técnica utiliza microondas de baja energía, que son absorbidas por los tejidos de manera diferente según su composición, densidad, microestructura, contenido de agua, entre otras características. Además, la tomografía por microondas ofrece ventajas prácticas y económicas, al ser una técnica más

[1]El término *dieléctrico* se refiere a un material que no conduce electricidad, pero puede ser polarizado por un campo eléctrico.

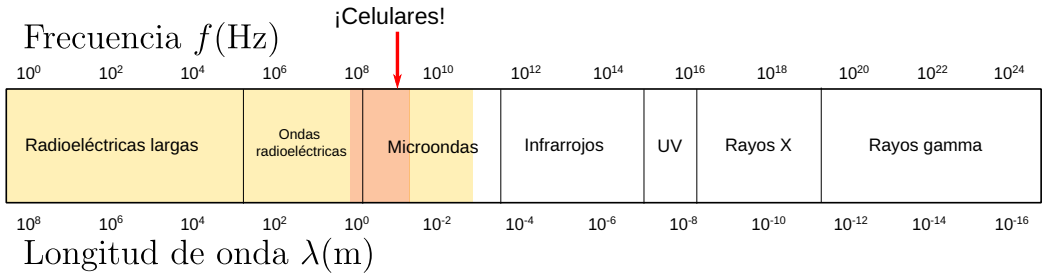

Figura 1: Espectro electromagnético. En rosa se indica el intervalo de dicho espectro correspondiente a las microondas.

accesible y de menor costo en comparación con otras modalidades de imagenología médica avanzada. Esto democratiza el acceso a diagnósticos de alta calidad, lo que es crucial para la detección temprana y el tratamiento efectivo de enfermedades óseas.

3.3. Objetivos de la PPS

El objetivo general de esta PPS es obtener modelos tipo CAD (de sus siglas en inglés Computer Aided Design) en dos dimensiones de manera semiautomática para la simulación intensiva de escenarios inspirados en imágenes médicas reales. El objetivo particular busca construir modelos de cortes de muñeca y/o tobillo para la simulación de mediciones de tomografía de microondas. El proceso consiste en la segmentación de imágenes, la generación de mallas para su simulación con el Método de Elementos Finitos (FEM de sus siglas en inglés Finite Element Method), su conversión a mallas regulares para ser usadas con métodos de dominio del tiempo (FDTD de sus siglas en inglés Finite Difference Time Domain) y, finalmente, el uso de estas simulaciones e inteligencia artificial para la reconstrucción de imágenes a partir de mediciones tomográficas. Remarcamos que para el entrenamiento supervisado de esta inteligencia artificial es necesario la utilización de grandes volúmenes de datos. En este trabajo hacemos énfasis en la generación semiautomática de modelos CAD[2] (por ejemplo: en la Fig. 2 se muestra un modelo CAD de un corte de muñeca). Las herramientas desarrolladas se utilizarán para la construcción de una base de datos y un software dedicado a la tomografía de microondas.

[2]En un modelo CAD se tienen parametrizadas las curvas que describen los objetos, de esta manera se puede generar mallas con una resolución arbitraria

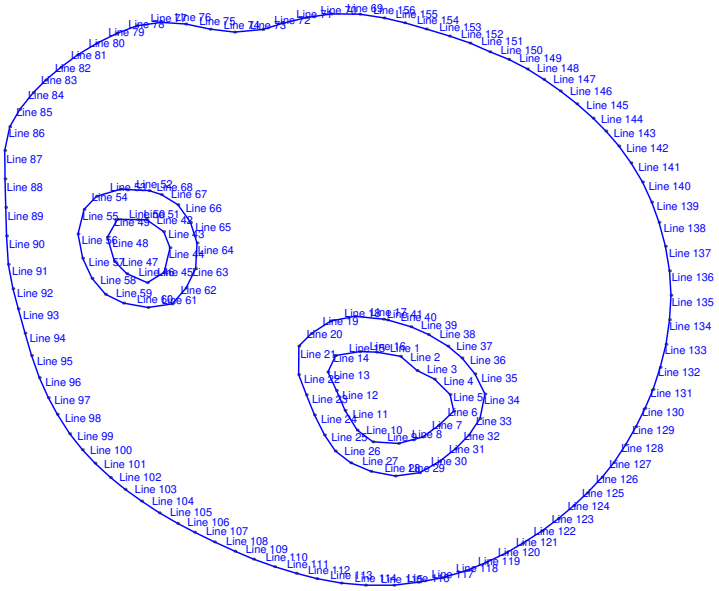

Figura 2: Se muestra la visualización de un archivo CAD con el software Gmsh [5]. , Debe observarse que se conocen y se tiene descripción parametrizada de cada punto, curva, y superficie del corte. Esto es importante a la hora de generar las mallas con resolución arbitraria.

3.4. Descripción de la creación de un generador de modelos intensivo (justificación)

La elección de enfocarse en el Método de Elementos Finitos en este proyecto se basa en su relevancia y eficacia probada en el ámbito de la tomografía por microondas [6]. El FEM es una técnica numérica ampliamente utilizada para resolver ecuaciones en derivadas parciales en problemas de ingeniería y física, lo que lo convierte en una herramienta idónea para abordar una simulación en la tomografía de microondas. Esta es la razón por la cual las mallas generadas en esta PPS son aptas para simulaciones con FEM.

La comparación con otros métodos numéricos, como el Método de los Momentos y el Método de Diferencias Finitas en el Dominio del Tiempo (FDTD), es de gran importancia para evaluar las fortalezas y limitaciones de cada enfoque en un contexto específico. Cada método tiene sus propias ventajas y desafíos, y comprender sus diferencias y aplicaciones respectivas proporciona una visión más completa y precisa de cómo abordar problemas particulares en este campo. En consecuencia, en esta PPS también desarrollaremos código que permita convertir mallas específicas para algoritmos de FEM a mallas utilizables con otros algoritmos (por ejemplo: FDTD).

4. Marco Teórico

4.1. Explicación de la tomografía de microondas

La tomografía de microondas es una técnica de imágenes no invasiva que utiliza ondas electromagnéticas de frecuencias comprendidas entre aproximadamente 100 MHz y 20 GHz. Se basa en la detección de cambios en las propiedades dieléctricas del objeto en estudio (ver Fig.3). En la TMO se busca reconstruir dicha distribución, "iluminando" el objeto (o tejido) dispersor con microondas y detectándolas con cierto número de antenas que pueden ser de distintos tipos y estar dispuestas en variadas configuraciones sobre un determinado Dominio de Medición. La distribución espacial de las propiedades dieléctricas se reconstruye en cierto dominio, denominado Dominio de Investigación. Las ondas electromagnéticas emitidas por los transmisores se propagan a través del objeto en estudio, y las variaciones en las propiedades dieléctricas (contraste dieléctrico) del tejido provocan dispersiones en las ondas. Este contraste se convierte en la base para la formación de imágenes, lo que permite obtener información sobre la estructura interna de los tejidos. Este enfoque ha despertado un interés significativo en la última década debido a su capacidad para generar imágenes médicas de manera no invasiva y con costos más bajos en comparación con otras técnicas. Además, la tomografía de microondas ha mostrado avances notables en la detección de enfermedades como el cáncer de mama [7] y hemorragias cerebrales [8], lo que destaca su potencial en aplicaciones médicas. En esta PPS nos centramos en construir de manera intensiva las geometrías inspiradas en modelos realistas que luego se podrán simular, tanto con FEM como con FDTD.

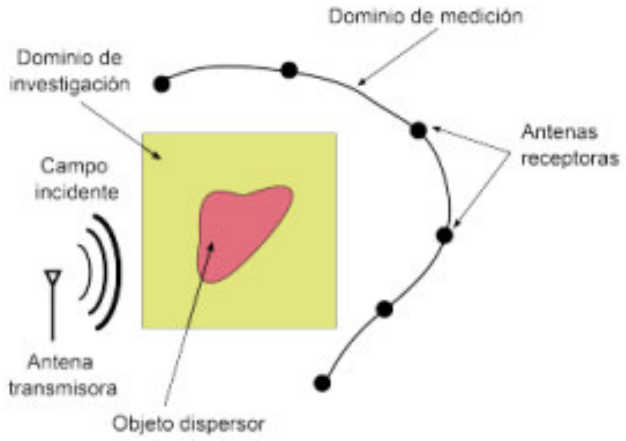

Figura 3: Esquema de medición en tomografía por microondas.

4.2. Descripción de las propiedades dieléctricas y su relevancia en la tomografía de microondas

Las propiedades dieléctricas se refieren a la respuesta de un material a la aplicación de un campo eléctrico. Comprender estas propiedades es esencial para interpretar las interacciones de las ondas electromagnéticas con los tejidos y, por ende, para el éxito de la tomografía de microondas como técnica de imágenes médicas. Las propiedades dieléctricas principales son la permitividad relativa (ε_r) y la conductividad (σ). La permitividad relativa es una medida de la capacidad de un material para polarizarse en respuesta a un campo eléctrico externo. Se expresa como la relación entre la permitividad del material y la permitividad del vacío (ε_0). Por otro lado, la conductividad describe la capacidad de un material para conducir corriente eléctrica cuando se somete a un campo eléctrico. Las propiedades dieléctricas dependen fuertemente de la frecuencia.

En el contexto de la tomografía de microondas, estas propiedades son cruciales ya que la interacción de las ondas electromagnéticas con los tejidos está directamente relacionada con el cambio de la permitividad relativa y la conductividad en el espacio. Por ejemplo, el músculo y el hueso cortical presentan diferentes valores de permitividad relativa y conductividad, lo que provoca cambios en la propagación de las ondas electromagnéticas. Estas variaciones se traducen en contrastes en las imágenes obtenidas mediante la tomografía de microondas, permitiendo así la visualización de estructuras internas y la detección de anomalías. En particular, se ha observado la variación de las propiedades dieléctricas de tejido óseo trabecular humano y su correlación con propiedades de microoestructura [9]. Esto, implica que con TMO se podría observar este cambio y de esta manera evaluar el estado de salud del tejido.

4.3. Descripción de los datos y metadatos proveniente de imágenes médicas

Como en esta PPS construiremos modelos inspirados en imágenes médicas (por resonancia o por tomografía computada), será necesario trabajar con datos provenientes de equipos clínicos. La mayoría de estos utilizan datos tipo DICOM. Este tipo de archivo, de sus siglas en inglés Digital Imaging and Communications in Medicine, es un estándar de transmisión de imágenes médicas y datos entre hardware de propósito médico. Las aplicaciones más comunes de este estándar son la visualización, almacenamiento, impresión y transmisión de las imágenes. El protocolo incluye la definición de un formato de fichero, un protocolo de comunicación de red basado en TCP/IP. En la

Figura 4, se muestra una representación de un conjunto de archivos DICOM en 3DSlicer.

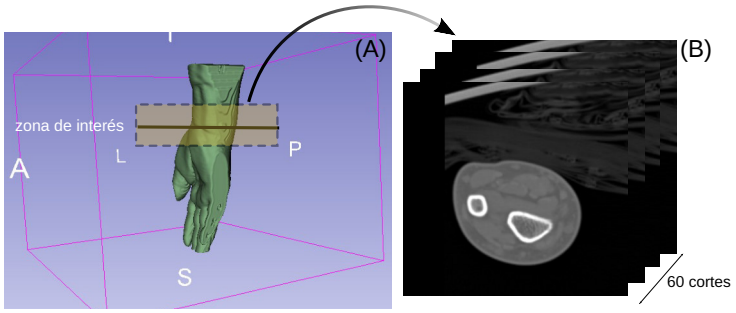

Figura 4: A. Representación en 3D de un modelo CT utilizando 3DSlicer, destacando la zona de interés. B. Cortes transversales obtenidos, a partir de los cuales se generarán los modelos en esta PPS.

El formato universal para imágenes médicas permite la interoperabilidad entre diferentes sistemas y equipos médicos. Además, tiene la capacidad de almacenar metadatos junto a la imagen, lo que facilita el soporte para múltiples modalidades de imagen como rayos X, resonancia magnética y tomografía.

En la medicina moderna, esto es fundamental porque facilita diagnósticos precisos y permite compartir información médica entre especialistas. También estandariza el manejo de imágenes diagnósticas y apoya la investigación médica avanzada.

La estructura técnica de este sistema incluye una cabecera de metadatos que contiene información detallada del paciente, como nombre, ID, fecha de nacimiento y parámetros de adquisición de imagen. Esto permite la identificación única de cada estudio médico. Por otro lado, los datos de imagen son una representación binaria de la imagen médica que soporta múltiples formatos y profundidades de bits, manteniendo una alta fidelidad diagnóstica y permitiendo compresión con pérdida mínima de información.

Entre las características técnicas avanzadas, se incluye el uso de etiquetas estandarizadas (tags), soporte para comunicación en red mediante Dicom networking y compatibilidad con sistemas PACS (Picture Archiving and Communication Systems). Además, incorpora mecanismos integrados de seguridad y privacidad para garantizar la protección de los datos médicos.

Para analizar los metadatos de un archivo Dicom, se puede utilizar la biblioteca pydicom [3]:

```
pydicom.dcmread(archivo.dcm)
```

[3] https://pydicom.github.io/

Dos datos cruciales para proyectos de procesamiento de imágenes médicas son el espaciado de píxeles (pixel spacing): permite convertir coordenadas de píxeles a unidades físicas (metros).

```
(0028, 0030) Pixel Spacing                              DS:
   [0.421488298, 0.421488298]
```

y el grosor del corte(Slice thickness): proporciona el espesor de la imagen en milímetros.

```
(0018, 0050) Slice Thickness                            DS: '
   2.11292686'
```

4.4. Segmentación de imágenes

La segmentación de imágenes es un proceso esencial en el análisis y procesamiento de imágenes, particularmente en el estudio de estructuras anatómicas complejas, como los huesos. Su principal objetivo es identificar y separar diferentes regiones o componentes dentro de una imagen para facilitar su análisis, modelado o manipulación.

Este proceso permite distinguir con precisión regiones específicas que representan de manera fiel las propiedades geométricas y estructurales de las áreas de interés. Además, la segmentación posibilita un filtrado más preciso mediante la identificación de patrones característicos, como variaciones en la densidad y composición de los tejidos.

La segmentación simplifica la interpretación de las imágenes, destacando características clave que pueden estar ocultas o ser difíciles de identificar en imágenes complejas. Esto es especialmente relevante en aplicaciones como la planificación quirúrgica, el diseño de prótesis, el diagnóstico médico, el análisis biomecánico, y nuestra aplicación particular modelado numérico computacional, donde una representación clara y precisa de las estructuras anatómicas resulta indispensable.

4.5. Descripción del tipo de herramienta CAD

Las herramientas CAD (Diseño Asistido por Computadora, por sus siglas en inglés) son softwares diseñados para facilitar la creación, modificación, análisis y optimización de diseños digitales. Estas herramientas permiten representar geometrías de manera precisa mediante el uso de modelos matemáticos, lo que resulta esencial en una amplia gama de aplicaciones de ingeniería y diseño. Una de las principales ventajas de los modelos CAD es

que, al proporcionar información matemática detallada sobre las curvas y superficies que describen una geometría, es posible generar mallas con cualquier resolución requerida para simulaciones numéricas o análisis estructurales.

En este trabajo, utilizaremos la herramienta Gmsh [5], una plataforma de código abierto ampliamente utilizada para la generación de mallas y la visualización de modelos CAD. Gmsh será explicado con más detalle en la sección 5.2, donde se describirán sus características y funcionalidades principales.

Como referencia adicional sobre el uso de herramientas CAD en el contexto de modelado y simulación, se puede consultar [10].

Archivo .geo

Para construir modelos CAD utilizando Gmsh, se pueden emplear archivos con extensión .geo que incorporan funciones específicas para la definición geométrica. Las principales funciones para este propósito son: *Point*, *Line*, *Line Loop* y *Plane Surface*.

Point

Un **Point** es una entidad geométrica que representa una posición en el espacio. En coordenadas cartesianas tridimensionales, un punto se define como:

$$P = (x, y, z)$$

Donde x, y, y z son las coordenadas del punto. Un parámetro extra en la definición del punto es el **gridsize**, que determina el tamaño de la malla o rejilla utilizada para discretizar un dominio geométrico. En simulaciones de elementos finitos, esta métrica es fundamental para establecer la densidad de nodos en el espacio.

Line

Una **Line** es un segmento que conecta dos puntos en el espacio. Si los puntos extremos de la línea son $P_1 = (x_1, y_1, z_1)$ y $P_2 = (x_2, y_2, z_2)$, entonces la línea puede representarse como:

$$L(t) = P_1 + t(P_2 - P_1), \quad t \in [0, 1]$$

aunque también puede tener otras funciones más complejas, como splines, por ejemplo.

Line Loop

Un **Line Loop** es una colección de líneas conectadas de manera secuencial que forman un contorno cerrado. Matemáticamente, esto implica que el punto final de la última línea coincide con el punto inicial de la primera.

$$L_1, L_2, \ldots, Ln \quad \text{con} \quad \text{inicio}(Li + 1) = \text{fin}(L_i), \quad \text{y} \quad \text{fin}(L_n) = \text{inicio}(L_1)$$

Plane Surface

Una **Plane Surface** es un área delimitada por un *Line Loop* en un plano bidimensional. La superficie puede definirse como el conjunto de puntos (x, y, z) que satisfacen:

$$\text{Superficie} = \{(x, y, z) \mid \text{interior del Line Loop}\}$$

Ejemplo

En este apartado se presenta un ejemplo práctico para la generación de un rectángulo en GMSH. A continuación, se muestra el código correspondiente:

Listing 1: Script para la creación de un rectángulo en GMSH

```
//Malla ejemplo
gridsize = .1;

Point(159) = {-0.125,-0.125,0,gridsize};
Point(160) = {-0.125,0.125,0,gridsize};
Point(161) = {0.125,0.125,0,gridsize};
Point(162) = {0.125,-0.125,0,gridsize};
Line(159) = {159,160};
Line(160) = {160,161};
Line(161) = {161,162};
Line(162) = {162,159};
Line Loop(6) = {159,160,161,162};
Plane Surface(7) = {6};
```

En la Figura 5, se puede observar el resultado de la malla generada a partir del script presentado.

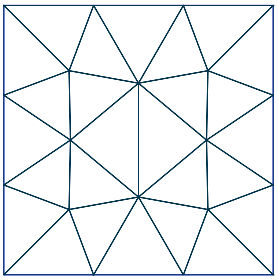

Figura 5: Resultado de la generación de mallas en un rectángulo utilizando GMSH.

5. Desarrollo Tecnológico

En este capítulo se abordará el desarrollo tecnológico aplicado a la segmentación de imágenes médicas. Se analizarán técnicas avanzadas como Marching Squares y Marching Cubes, utilizadas para identificar y delinear estructuras anatómicas específicas. Se presentará el ecosistema de herramientas de Python empleado, como NumPy, PyDiCOM, scikit-image y SciPy, las cuales facilitan el análisis espacial, las transformaciones geométricas y la creación de representaciones volumétricas precisas. Finalmente, se describirá cómo la implementación del software sigue los principios de diseño SOLID, asegurando modularidad, mantenibilidad y escalabilidad.

5.1. Algoritmos de segmentación

Los algoritmos de segmentación son técnicas empleadas para dividir conjuntos de datos, imágenes o señales en regiones donde los elementos comparten propiedades similares. Esta segmentación es fundamental para identificar patrones, estructuras o características específicas dentro de los datos. En particular, nosotros estamos interesados en detectar contornos de hueso cortical, hueso trabecular y tejidos blandos.

Para este proyecto utilizamos la herramienta skimage [4]. En particular, como algoritmo de segmentación, un ejemplo destacado es el algoritmo Marching Squares, diseñado para identificar contornos en una cuadrícula bidimensional. Este método examina las esquinas de cada celda cuadrada de la cuadrícula, determinando cómo se conectan los bordes para delinear el contorno de las regiones de interés. En la Figura 6, se presenta un ejemplo del algoritmo Marching Squares, donde el umbral fue configurado en 5 unidades, resultando en los contornos mostrados.

En ciertos escenarios, el algoritmo puede encontrar configuraciones donde no está claro cómo conectar los puntos del contorno. Estos son conocidos como casos ambiguos. Aunque se mencionan principalmente en el contexto bidimensional, este concepto resulta útil para entender problemas similares que surgen al trabajar con algoritmos en tres dimensiones.

El algoritmo Marching Cubes expande el concepto de Marching Squares al espacio tridimensional (3D). Es ampliamente utilizado para representar superficies a partir de datos volumétricos, buscando iso-superficies que conecten puntos con un mismo valor (por ejemplo, temperatura, densidad u otra propiedad) dentro de un volumen. El algoritmo analiza pequeños cubos dentro de un volumen y clasifica las esquinas de cada cubo según si están por

[4] `https://scikit-image.org/`

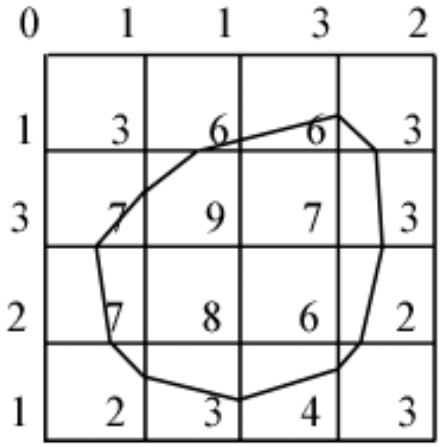

Figura 6: Ejemplo del algoritmo Marching Squares.La pendiente de las líneas depende del peso de los nodos. Fuente: `https://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html`.[11]

encima o por debajo de un valor umbral. Posteriormente, genera triángulos para construir la superficie. Sin embargo, en el espacio tridimensional también surgen casos ambiguos, donde no está claro cómo deberían conectarse las superficies. Estos casos representan desafíos específicos y requieren estrategias adicionales para resolverse. Una solución típica implica el uso de tablas de configuración mejoradas o heurísticas basadas en propiedades geométricas.

5.2. Descripción de las bibliotecas y herramientas de Python utilizadas en el proyecto

Python cuenta con muchas bibliotecas hechas por la comunidad que ayudan a poder realizar grandes proyectos. En este caso se utilizaron las siguientes:

os Es una biblioteca que permite interactuar con el sistema operativo, útil para manejar archivos y directorios.

numpy Es una biblioteca para cálculo numérico y manejo de arreglos multidimensionales, utilizada para procesar datos y realizar cálculos matemáticos.

matplotlib.pyplot Herramienta para la creación de gráficos y visualizaciones, empleada para graficar resultados y analizar datos visualmente.

pydicom Biblioteca para trabajar con imágenes médicas en formato Dicom, permite la lectura, manipulación y extracción de información de imágenes médicas.

skimage.measure Submódulo de `skimage` para analizar y medir propiedades de imágenes, utilizado en el procesamiento y análisis de características.

meshio Biblioteca para leer y escribir mallas en múltiples formatos, empleada para manipular estructuras tridimensionales.

scipy.spatial.KDTree Estructura eficiente para búsqueda de vecinos más cercanos en espacios multidimensionales.

scipy.spatial.ConvexHull Herramienta para calcular la envolvente convexa de un conjunto de puntos.

shapely.geometry.Polygon Herramientas para crear y manipular geometrías 2D, utilizadas para trabajar con polígonos y operaciones geométricas.

5.3. Integración de buenas prácticas de desarrollo de software

Se siguieron los principios SOLID introducidos por Robert C. Martin, los cuales son un conjunto de buenas prácticas de programación orientada a objetos. Estos principios ayudan a desarrollar sistemas que sean modulares, fáciles de mantener y comprender, y que favorezcan la escalabilidad del software. A continuación, se explican las siglas que conforman SOLID:

- **S: Single Responsibility Principle (Principio de responsabilidad única)**. Cada clase debe tener una única responsabilidad o razón para cambiar. Esto facilita la identificación y aislamiento de funcionalidades.
- **O: Open/Closed Principle (Principio de abierto/cerrado)**. Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para extensión, pero cerradas para modificación, permitiendo que nuevas funcionalidades se añadan sin alterar el código existente.
- **L: Liskov Substitution Principle (Principio de sustitución de Liskov)**. Los objetos de una clase derivada deben poder reemplazar a objetos de su clase base sin alterar el comportamiento del programa.

- **I: Interface Segregation Principle (Principio de segregación de interfaces)**. Una clase no debería estar obligada a implementar interfaces que no utiliza. Es preferible crear interfaces específicas para cada propósito.
- **D: Dependency Inversion Principle (Principio de inversión de dependencias)**. Los módulos de alto nivel no deben depender de módulos de bajo nivel; ambos deben depender de abstracciones. Asimismo, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones.

Al aplicar estos principios, se logró que el código escrito fuera más modular, fácil de mantener y comprender, reduciendo la complejidad y aumentando la robustez del sistema.

6. Resultados

En este capítulo, discutimos la conversión de archivos DICOM en un formato GEO mediante un script automatizado. Se describe un flujo de trabajo que incluye el procesamiento de archivos DICOM, la detección de contornos de huesos y piel con diferentes umbrales, su filtrado y ordenamiento, y la posterior generación de archivos `.geo` compatibles con Gmsh. El script principal, `procimage2gmsh_vdb.py`, utiliza bibliotecas como `numpy`, `matplotlib`, y `pydicom`, ofreciendo un procesamiento por lotes eficiente.

Además, se detalla el modo de uso, incluyendo la instalación de dependencias y la ejecución con parámetros como directorios de entrada (`-i`) y salida (`-o`), lo que permite automatizar la generación de geometrías para simulaciones numéricas. Posteriormente, se emplea Gmsh para convertir los archivos `.geo` generados en archivos `.msh`, adecuados para simulaciones numéricas basadas en elementos finitos.

Finalmente, un script adicional convierte las mallas generadas, permitiendo transformar las simulaciones de FEM a FDTD y genera un archivo específico requerido para este propósito. Un último script toma este archivo y produce una visualización en formato `.png`, facilitando la interpretación gráfica de los resultados. Todo el flujo de trabajo, incluyendo los scripts y las instrucciones detalladas, está disponible en el repositorio público `GitHub` en la siguiente dirección:

`https://github.com/matiasbenary/procesador_image_pps`

6.1. Conversión de DICOM a GEO

6.1.1. Diagrama de flujo general

El la Fig. 7 se muestra el diagrama de flujo que ilustra el funcionamiento general del script utilizado para la conversión de archivos Dicom a formato GEO.

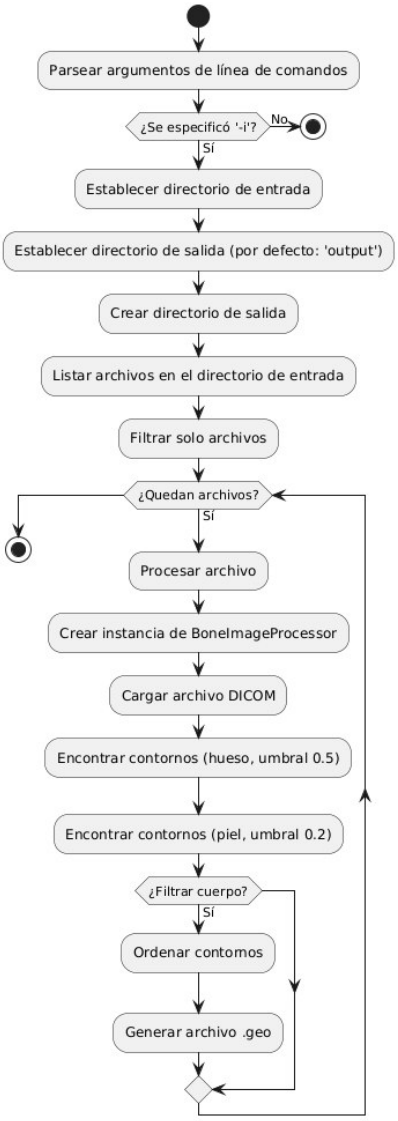

Figura 7: Diagrama de flujo del proceso de conversión de Dicom a GEO.

6.1.2. Script Principal

El núcleo del procesamiento se encuentra en el script `procimage2gmsh_vdb.py`. A continuación, se presenta un fragmento representativo del código:

```
def main():
    parser = argparse.ArgumentParser(description="Procesa
        im genes DICOM para generar archivos .geo")
    parser.add_argument('-i', '--input', required=True,
```

```
                        help='Directorio de entrada que
                            contiene archivos DICOM')
    parser.add_argument('-o', '--output', default='output',
                        help='Directorio de salida para los
                            archivos procesados')

    args = parser.parse_args()

    os.makedirs(args.output, exist_ok=True)

    filename = os.listdir(args.input)
    only_files = [f for f in filename if os.path.isfile(os.
        path.join(args.input, f))]

    for filename in only_files:
        processor = BoneImageProcessor(os.path.join(args.
            input, filename))

        processor.load_dicom()

        processor.find_contours(0.5, 'bone')
        processor.find_contours(0.2, 'skin')

        if processor.filter_body():
            # Descomentar para ver las imagenes con los
                contornos
            # processor.plot_final_contours()
            processor.order_contours()
            processor.generate_gmsh_geo(os.path.join(args.
                output, f'munieca-{filename[:-4]}.geo'))
```

En este código, se utiliza la función find_contours con un umbral de 0.5 para identificar tejido óseo y 0.2 para la piel. Este algoritmo implementa el método de Marching Squares, previamente descrito.

6.1.3. Modo de Uso

1. **Instalar dependencias:** Ejecute el siguiente comando para instalar los paquetes requeridos:

```
pip install -r requirements.txt
```

2. **Ejecutar el script:** Ejecute el script con el siguiente comando:

```
python3 procimage2gmsh_vdb.py -i data/input2 -o
    output
```

3. **Descripción de los parámetros:**

- `-i` (`--input`): Directorio que contiene los archivos Dicom de entrada (obligatorio).
- `-o` (`--output`): Directorio donde se guardarán los archivos procesados. Si no se especifica, el valor predeterminado es `output`.

6.1.4. Detalles del Script

El flujo principal del programa está contenido en la función `main`. A continuación, se detalla el proceso:

- Se parsean los argumentos de entrada con `argparse`:

```
parser = argparse.ArgumentParser(description="Procesa
    im genes DICOM para generar archivos .geo")
parser.add_argument('-i', '--input', required=True,
                    help='Directorio de entrada que
                        contiene archivos DICOM')
parser.add_argument('-o', '--output', default='output
    ',
                    help='Directorio de salida para
                        los archivos procesados')

args = parser.parse_args()
```

- Se crean las carpetas de salida necesarias usando `os.makedirs`.

```
os.makedirs(args.output, exist_ok=True)
```

- Se obtienen los archivos Dicom y se procesan iterativamente:

```
filename = os.listdir(args.input)
only_files = [f for f in filename if os.path.isfile(
   os.path.join(args.input, f))]

for filename in only_files:
```

A su vez, en cada iteración se realiza lo siguiente:

1. Se leen las imágenes y se procesan con la clase `BoneImageProcessor`, más adelante se verá en detalle esta clase. Ingresando como dato el path del archivo Dicom
2. Se extraen los contornos de hueso y piel utilizando `find_contours`. Esta función necesita, como parámetro, el threshold y la etiqueta.
3. Se aplica un filtro de validación con `filter_body`.

4. Los contornos se ordenan con `order_contours`.
5. Finalmente, se genera un archivo `.geo` compatible con Gmsh.

```
        processor = BoneImageProcessor(os.path.join(args.
            input, filename))

        processor.load_dicom()

        processor.find_contours(0.5, 'bone')
        processor.find_contours(0.2, 'skin')

        if processor.filter_body():
            # Descomentar para ver las imagenes con los
                contornos
            # processor.plot_final_contours()
            processor.order_contours()
            processor.generate_gmsh_geo(os.path.join(args
                .output, f'munieca-{filename[:-4]}.geo'))
```

6.1.5. Nota

Para visualizar las imágenes con los contornos generados, descomente la línea correspondiente que invoca `plot_final_contours()` en el script. Si se activa esta función, el script queda a la espera de que se cierre la visualización. Es preferible que esté desactivada para que sea un procesamiento automático en lote.

6.2. Revisión de clase BoneImageProcessor

Este código implementa un sistema para procesar Dicom, detectar contornos, analizar su forma, y generar geometrías compatibles con el software `Gmsh`. En la Fig. 8, se describen los elementos en principales en formato de uml:

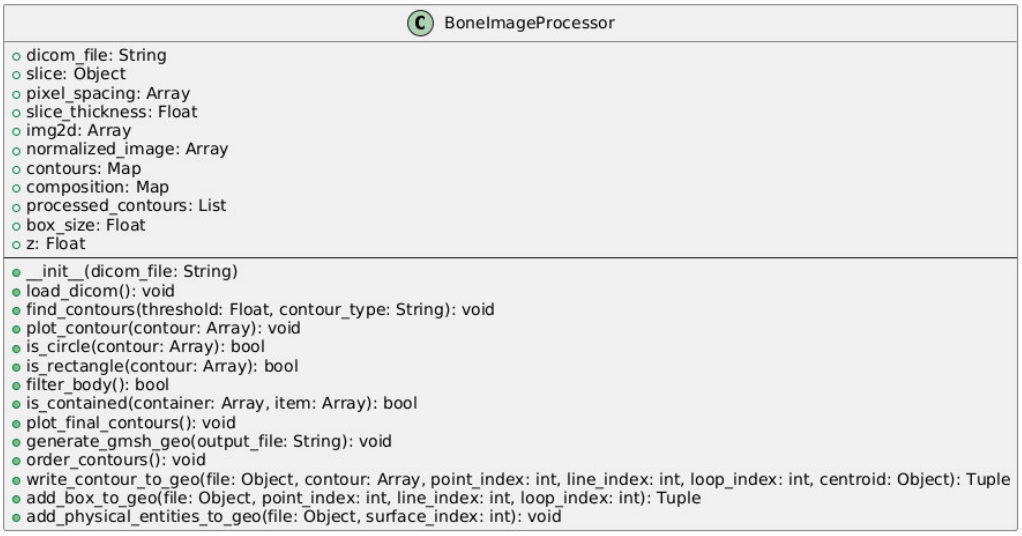

Figura 8: Uml de BoneImageProcessor

6.2.1. Importación de bibliotecas

El código utiliza varias bibliotecas de Python como `numpy`, `matplotlib`, y `pydicom`, que proporcionan funciones para procesamiento de imágenes, cálculo de métricas, y manipulación de archivos Dicom como se mencionó anteriormente.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import pydicom
from skimage import measure
from scipy.spatial import ConvexHull
from shapely.geometry import Polygon
```

6.2.2. Definición de funciones auxiliares

- `calculate_perimeter(contour)`: Calcula el perímetro de un contorno.
- `remove_duplicate_vectors(vectors)`: Elimina vectores duplicados de una lista.

```
def calculate_perimeter(contour):
    return np.sum(np.sqrt(np.sum(np.diff(contour, axis=0) **
        2, axis=1)))

def remove_duplicate_vectors(vectors):
```

```
    unique_vectors = set(tuple(vector) for vector in vectors)

    return [list(vector) for vector in unique_vectors]
```

6.2.3. Resumen de métodos

Sus métodos principales son:

- `load_dicom`: Carga el archivo Dicom y normaliza la imagen.
- `find_contours`: Encuentra los contornos en la imagen utilizando un umbral y una etiqueta.
- `plot_contour`: Grafica los contornos sobre la imagen original, se utiliza para depurar.
- `is_circle` e `is_rectangle`: Verifican la forma de los contornos.
- `filter_body`: Verifica que las formas de los contornos sean las esperadas. Se utiliza para determinar si los contornos son adecuados para poder generar el archivo `.geo` para Gmsh.
- `is_contained`: Verifica que una geometría esté dentro de otra.
- `plot_final_contours`: Abre una ventana con la imagen y el contorno señalado. Esto requiere una acción manual y debería usarse para realizar un control.
- `generate_gmsh_geo`: Genera un archivo `.geo` para Gmsh.
- `order_contours`: Ordena los contornos para que las capas generadas sean correctas y estén en el mismo nivel.
- `write_contour_to_geo`: Agrega un contorno al archivo `.geo`.
- `add_box_to_geo`: Agrega el marco al archivo `.geo`.
- `add_physical_entities_to_geo`: Agrega las superficies físicas al archivo `.geo`. Es la última función que se ejecuta.

6.2.4. Variables del constructor

A continuación se explicará la finalidad de cada variable utilizada por la clase:

- `dicom_file`: Almacena la ruta o referencia al archivo Dicom que contiene la imagen médica y sus metadatos.
- `slice`: Contendrá el objeto Dicom cargado, que incluye tanto la imagen como los metadatos.
- `pixel_spacing`: Representa la distancia física entre los píxeles en el plano de la imagen, obtenida de los metadatos Dicom.
- `slice_thickness`: Almacena el grosor del corte de la imagen en milímetros, también derivado de los metadatos Dicom.
- `img2d`: Contiene la matriz bidimensional de los datos de píxeles de la imagen Dicom.
- `normalized_image`: Una versión normalizada de `img2d`, con valores de píxeles escalados entre 0 y 1.
- `contours`: Un diccionario que almacena contornos detectados en la imagen, categorizados en dos tipos:
  - `"bone"`: Contornos relacionados con huesos.
  - `"skin"`: Contornos relacionados con la piel.
- `composition`: Un diccionario que organiza los contornos detectados según su relación jerárquica, similar a `contours`.
- `processed_contours`: Una lista que almacenará contornos que han sido procesados para exportación o visualización.
- `box_size`: Define el tamaño (en metros) de un cuadro utilizado en operaciones como mallado (*meshing*).
- `z`: Representa la coordenada z de la posición de la imagen en el espacio del paciente, extraída de los metadatos Dicom.

6.2.5. Método `load_dicom`

La función `load_dicom` se encarga de cargar un archivo Dicom y extraer tanto los datos de la imagen como los metadatos relevantes. Su propósito es inicializar las variables clave de la clase para que puedan ser utilizadas en etapas posteriores de procesamiento.

```
    def load_dicom(self):
        self.slice = pydicom.dcmread(self.dicom_file)
        self.pixel_spacing = self.slice.PixelSpacing
        self.slice_thickness = self.slice.SliceThickness
        x, y, self.z = self.slice.ImagePositionPatient
        self.img2d = self.slice.pixel_array
        self.normalized_image = self.img2d / np.amax(self.
            img2d)
```

Descripción de las operaciones

- `self.slice = pydicom.dcmread(self.dicom_file)`: Carga el archivo Dicom especificado y lo almacena en `self.slice`.
- `self.pixel_spacing = self.slice.PixelSpacing`: Extrae la información de la distancia entre píxeles desde los metadatos del archivo Dicom.
- `self.slice_thickness = self.slice.SliceThickness`: Obtiene el grosor del corte de la imagen desde los metadatos.
- `x, y, self.z = self.slice.ImagePositionPatient`: Almacena la posición del origen de la imagen en las coordenadas del paciente, incluyendo la posición z.
- `self.img2d = self.slice.pixel_array`: Almacena los datos de píxeles en una matriz bidimensional.
- `self.normalized_image = self.img2d / np.amax(self.img2d)`: Normaliza los valores de los píxeles dividiendo cada valor por el máximo de la matriz, escalándolos entre 0 y 1.

6.2.6. Método `find_contours`

La función `find_contours` detecta contornos en una imagen normalizada (`self.normalized_image`) basada en un umbral específico. Los contornos encontrados se filtran según su tamaño y forma, y luego se almacenan en el diccionario `self.contours` bajo una categoría específica (`contour_type`).

```
def find_contours(self, threshold, contour_type):
    contours = measure.find_contours(self.
       normalized_image, threshold)
    self.contours[contour_type] = [
        c for c in contours if len(c) > 30 and not self.
           is_rectangle(c)
    ]
```

Descripción de los parámetros

- `threshold`: Un valor numérico que define el umbral de intensidad para detectar los contornos. Las regiones en la imagen con valores iguales o mayores a este umbral son evaluadas.
- `contour_type`: Una cadena de texto (`"bone"` o `"skin"`, en este caso particular) que especifica el tipo de contorno que se está buscando. Los contornos detectados se almacenan en el diccionario `self.contours` bajo esta clave.

Flujo del Método

1. Utiliza `measure.find_contours` de la biblioteca `scikit-image` para detectar los contornos en la imagen `self.normalized_image` con el umbral dado.
2. Filtra los contornos según dos criterios:
   - La longitud del contorno (`len(c) >30`): Solo se consideran contornos con más de 30 puntos para eliminar aquellos que son demasiado pequeños.
   - La forma del contorno (`not self.is_rectangle(c)`): Excluye contornos que tienen una forma rectangular, utilizando la función auxiliar `is_rectangle`[5].
3. Los contornos resultantes se almacenan en el diccionario `self.contours` bajo la clave `contour_type`.

[5]Esta función se diseñó para evitar que los objetos que se colocan para que el paciente apoye el brazo o el pie se excluyan del modelo.Además si el paciente presenta alguna irregularidad

6.3. Método `plot_contour`

La función `plot_contour` se encarga de mostrar un contorno específico sobre una imagen normalizada. Utiliza la biblioteca `matplotlib` para crear un gráfico en el que se superpone el contorno sobre la imagen.

```
    def plot_contour(self, contour):
        fig, ax = plt.subplots(1, 1, figsize=(8, 8))
        ax.imshow(self.normalized_image, cmap=plt.cm.gray,
            interpolation="nearest")
        ax.plot(contour[:, 1], contour[:, 0], linewidth=2)
        # plt.show()
```

Descripción de los parámetros

- `contour`: Un arreglo de `numpy` que contiene las coordenadas de un contorno, con cada punto representado por un par de coordenadas `(x, y)`.

Flujo del Método

1. Se crea una figura y un eje de gráficos utilizando `plt.subplots` con un tamaño de figura de `8x8` pulgadas.

2. La imagen normalizada (`self.normalized_image`) se visualiza en el gráfico con un mapa de colores en escala de grises (`cmap=plt.cm.gray`) y sin suavizado de píxeles (`interpolation="nearest"`).

3. Se superpone el contorno sobre la imagen utilizando `ax.plot`, donde las coordenadas del contorno se pasan como `contour[:, 1]` para las coordenadas `y` (columnas) y `contour[:, 0]` para las coordenadas `x` (filas). Además, se establece el grosor de la línea del contorno con `linewidth=2`.

4. Finalmente, `plt.show()` se llama para mostrar la imagen con el contorno sobrepuesto en una ventana gráfica.

6.4. Método `is_circle`

La función `is_circle` se utiliza para determinar si un contorno dado tiene una forma aproximada a un círculo. Esta evaluación se realiza mediante el cálculo de la *circularidad*, que es una medida que compara el área del contorno con su perímetro. Si el contorno tiene una forma que se aproxima a un círculo, la función devuelve `True`, de lo contrario, devuelve `False`.

```
    def is_circle(self, contour):
        hull = ConvexHull(contour)
        area = hull.volume
        # area = measure.area(contour)
        perimeter = calculate_perimeter(contour)
        if perimeter == 0:
            return False
        circularity = (4 * np.pi * area) / (perimeter**2)
        return 0.7 <= circularity <= 1.3
```

Descripción de los parámetros

- `contour`: Un arreglo de `numpy` que contiene las coordenadas de un contorno, representado por una secuencia de puntos.

Flujo del Método

1. La función primero calcula el *envolvente convexa* (`ConvexHull(contour)`) del contorno utilizando el algoritmo de la envolvente convexa. Esto proporciona una forma convexa mínima que rodea el contorno original.

2. Se obtiene el *área* de la envolvente convexa con `hull.volume`.

3. Luego, la función calcula el *perímetro* del contorno utilizando la función `calculate_perimeter(contour)`. Si el perímetro es igual a cero, lo que indicaría un contorno inválido o degenerado, la función devuelve `False`.

4. La *circularidad* se calcula con la fórmula:

$$\text{circularidad} = \frac{4\pi \times \text{área}}{\text{perímetro}^2}$$

   Esta fórmula da un valor cercano a 1 para formas que son circulares. Cuanto más cercano a 1, más circular es el contorno.

5. Finalmente, la función devuelve `True` si la circularidad está dentro del rango de [0,7, 1,3], lo que indica que el contorno tiene una forma aproximadamente circular. Si no es así, devuelve `False`.

6.5. Método `is_rectangle`

La función `is_rectangle` determina si un contorno dado tiene una forma aproximada a un rectángulo. Esto se evalúa utilizando la relación entre las dimensiones del contorno (ancho y alto) y calculando la relación de aspecto

(ratio). Si la relación de aspecto es mayor a 5 o menor a 0.2, se considera que el contorno tiene una forma rectangular.

```
    def is_rectangle(self, contour):
        xmax, ymax = np.max(contour * self.pixel_spacing[0] *
            1.0e-3, axis=0)
        xmin, ymin = np.min(contour * self.pixel_spacing[0] *
            1.0e-3, axis=0)
        ratio = (xmax - xmin) / (ymax - ymin)
        return ratio > 5 or ratio < 0.2
```

Descripción de los parámetros

- `contour`: Un arreglo de `numpy` que contiene las coordenadas de un contorno, representado por una secuencia de puntos.

Flujo de la función

1. La función comienza multiplicando las coordenadas del contorno por `self.pixel_spacing[0]` y un factor de escala de $1{,}0e{-}3$, que convierte las unidades a milímetros. Este paso es importante si las coordenadas del contorno están en píxeles y se desea convertirlas a unidades físicas.

2. Luego, la función calcula las coordenadas máximas ($x_{\max}$, $y_{\max}$) y mínimas ($x_{\min}$, $y_{\min}$) en las dimensiones x y y del contorno, utilizando las funciones `np.max` y `np.min`, respectivamente.

3. Se calcula la relación de aspecto (ratio) del contorno, que es el cociente entre la diferencia en las coordenadas x ($x_{\max} - x_{\min}$) y la diferencia en las coordenadas y ($y_{\max} - y_{\min}$):

$$\text{ratio} = \frac{x_{\max} - x_{\min}}{y_{\max} - y_{\min}}$$

4. Finalmente, la función devuelve `True` si la relación de aspecto es mayor a 5 o menor a 0.2, lo que indica que el contorno tiene una forma rectangular, es decir, que es mucho más largo en una dimensión que en la otra.

6.6. Método `filter_body` y `is_contained`

Estos métodos están diseñados verificar que los contornos obtenidos son validos. La función `filter_body` agrupa los contornos de los huesos que están contenidos dentro de los contornos de la piel y organiza la composición de

los mismos, mientras que la función `is_contained` se usa para verificar si un contorno está completamente contenido dentro de otro contorno.

Método `filter_body`

El método `filter_body` recorre los contornos de la piel y los huesos, y selecciona aquellos huesos que están completamente contenidos dentro de los contornos de la piel. Si encuentra un grupo de cuatro huesos contenidos dentro de un contorno de piel, agrega ese contorno de piel y los huesos correspondientes a la composición.

```
    def filter_body(self):
        containers = [
            {
                "container": skin,
                "items": [
                    bone
                    for bone in self.contours["bone"]
                    if self.is_contained(skin, bone)
                ],
            }
            for skin in self.contours["skin"]
        ]

        for container in containers:
            if len(container["items"]) == 4:
                self.composition["skin"].append(container["
                    container"])
                self.composition["bone"].extend(container["
                    items"])
                return True
        return False
```

Descripción de los parámetros

- `self.contours["skin"]`: Contornos que representan la piel, almacenados en un diccionario.
- `self.contours["bone"]`: Contornos que representan los huesos, almacenados en un diccionario.
- `self.composition`: Un diccionario que contiene los contornos de piel y huesos seleccionados.

Flujo del método

1. Primero, la función crea una lista `containers`, que consiste en un diccionario para cada contorno de piel (`skin`). Para cada contorno de piel, se filtran los contornos de hueso que están contenidos dentro de él, usando la función `is_contained`.

2. Luego, la función recorre los `containers` generados y verifica si alguno contiene exactamente 4 huesos. Si es así, agrega el contorno de piel a `self.composition["skin"]` y los huesos a `self.composition["bone"]`, y retorna `True`.

3. Si no se encuentra ningún grupo de cuatro huesos dentro de la piel, la función retorna `False`.

Método `is_contained`

La función `is_contained` determina si un contorno de hueso está completamente contenido dentro de un contorno de piel. Esto se evalúa comparando las coordenadas mínimas y máximas de los contornos de la piel y los huesos.

```
    def is_contained(self, container, item):
        container_bounds = np.min(container, axis=0), np.max(
            container, axis=0)
        item_bounds = np.min(item, axis=0), np.max(item, axis
            =0)
        return (
            container_bounds[0][0] <= item_bounds[0][0]
            and container_bounds[0][1] <= item_bounds[0][1]
            and container_bounds[1][0] >= item_bounds[1][0]
            and container_bounds[1][1] >= item_bounds[1][1]
            and not np.array_equal(container_bounds,
                item_bounds)
        )
```

Descripción de los parámetros

- `container`: El contorno del contenedor (piel) que se está evaluando.

- `item`: El contorno del item (hueso) que se está verificando si está contenido dentro del contenedor.

Flujo del método

1. El método calcula los límites del contorno de la piel (`container`) y del hueso (`item`) usando `np.min` y `np.max` para obtener las coordenadas mínimas y máximas en las dimensiones x y y.

2. Luego, se compara si el contorno del hueso está completamente contenido dentro del contorno de la piel. Esto se evalúa comprobando si las coordenadas mínimas del hueso son mayores o iguales que las del contorno de la piel y si las coordenadas máximas del hueso son menores o iguales que las del contorno de la piel.

3. Además, se verifica que los límites del contorno de la piel no sean iguales a los del hueso, para asegurarse de que no estamos considerando contornos idénticos.

4. Si todas estas condiciones se cumplen, la función retorna `True`, indicando que el hueso está contenido dentro de la piel; de lo contrario, retorna `False`.

6.7. Método `plot_final_contours`

El método `plot_final_contours` se encarga de visualizar los contornos finales sobre la imagen 2D normalizada, mostrando tanto los contornos de la piel como los de los huesos. El método genera un gráfico en el cual se superponen los contornos finales, escalados y posicionados según las dimensiones físicas de la imagen.

```
    def plot_final_contours(self):
        fig, ax = plt.subplots(figsize=(8, 8))
        extent = [
            0,
            self.img2d.shape[1] * self.pixel_spacing[0] * 1.0
                e-3,
            0,
            self.img2d.shape[0] * self.pixel_spacing[0] * 1.0
                e-3,
        ]
        ax.imshow(
            self.normalized_image,
            cmap=plt.cm.gray,
            interpolation="nearest",
            extent=extent,
            origin="lower",
        )
```

```
        for contour_type, contours in self.composition.items
            ():
             for contour in contours:
                 ax.plot(
                     contour[:, 1] * self.pixel_spacing[0] *
                         1.0e-3,
                     contour[:, 0] * self.pixel_spacing[0] *
                         1.0e-3,
                     linewidth=2,
                 )

        ax.set_title("Final Contours")
        ax.set_xlabel("x (m)")
        ax.set_ylabel("y (m)")
        plt.show()
```

Descripción de los parámetros

- `self.normalized_image`: La imagen 2D normalizada que se va a mostrar.
- `self.img2d`: La imagen 2D original que contiene los píxeles a visualizar.
- `self.pixel_spacing`: El espaciado de los píxeles en la imagen, utilizado para convertir las coordenadas de píxeles a unidades físicas (metros).
- `self.composition`: Un diccionario que contiene los contornos finales clasificados (por ejemplo, "pielz "hueso").

Flujo de la función

1. Primero, se crea una figura (`fig`) y un eje (`ax`) usando `plt.subplots()` con un tamaño de 8x8 pulgadas.
2. Luego, se calcula el `extent` de la imagen para ajustar las coordenadas físicas en el gráfico. Se utiliza el tamaño de la imagen y el espaciado de los píxeles para calcular las dimensiones físicas en metros.
3. La imagen normalizada se muestra en el eje con la función `ax.imshow()`. Se especifica el mapa de colores `gray`, el tipo de interpolación `nearest`, y se ajusta el `extent` para que las coordenadas de los píxeles se correspondan con las dimensiones físicas.
4. Posteriormente, se recorre `self.composition`, que contiene los contornos clasificados. Para cada tipo de contorno (como "piel.º "hueso"),

se trazan las líneas de contorno sobre la imagen usando `ax.plot()`. Las coordenadas de los contornos se multiplican por el espaciado de los píxeles para convertirlas a unidades físicas en metros.

5. Finalmente, se ajustan los títulos y las etiquetas de los ejes, y se muestra el gráfico con `plt.show()`.

Propósito de la función

Esta función es útil para visualizar de manera clara y precisa los contornos finales de la piel y los huesos, superpuestos sobre la imagen original, escalados a unidades físicas. Permite comprobar la precisión y calidad de los contornos detectados y procesados en la imagen 2D.

6.8. Método `order_contours`

El método `order_contours` organiza y procesa los contornos de huesos y piel, clasificándolos en contornos internos y externos y asignándoles identificadores únicos. Los contornos internos se procesan de acuerdo a su relación espacial (contenedor y contenido), mientras que los contornos de la piel se consideran externos. El método utiliza un contador para asignar un ID único a cada contorno procesado.

```
    def order_contours(self):
        aux = []
        counter = 1
        skin = ""
        for container in self.composition["bone"]:
            for item in self.composition["bone"]:
                if max(container[:, 1]) != max(item[:, 1])
                   and self.is_contained(
                    container, item
                ):
                    self.processed_contours.append(
                        {
                            "item": item,
                            "surface": str(counter),
                            "type": "internal",
                            "id": counter,
                        }
                    )
                    counter += 1
                    aux.append(counter)
                    self.processed_contours.append(
                        {
                            "item": container,
```

```
                        "surface": str(counter) + "," +
                            str(counter - 1),
                        "type": "internal",
                        "id": counter,
                    }
                )
                skin = "," + str(counter)+skin
                counter += 1
    for contour in self.composition["skin"]:
        self.processed_contours.append(
            {
                "item": contour,
                "surface": str(counter) + skin,
                "type": "external",
                "id": counter,
            }
        )
```

Descripción de los parámetros

- `self.composition`: Un diccionario que contiene los contornos clasificados como "hueso" "piel". Cada tipo de contorno está representado por una lista de contornos.
- `self.processed_contours`: Una lista donde se almacenan los contornos procesados, con información adicional como el tipo y el ID.
- `aux`: Una lista auxiliar que almacena valores de contador para gestionar la relación entre contornos.
- `counter`: Un contador que se incrementa para asignar un ID único a cada contorno procesado.
- `skin`: Una cadena que almacena los identificadores de los contornos de la piel, utilizada para vincular los contornos internos con los externos.

Flujo de la función

1. Se inicializa la lista auxiliar `aux` y el contador `counter`, que comenzará en 1. La variable `skin` se inicializa como una cadena vacía.
2. La función recorre los contornos de hueso en `self.composition["bone"]`. Para cada contorno de hueso `container`, compara su posición con otros contornos `item` de hueso.

3. Si el contorno `item` está contenido dentro del contorno `container`, se agregan dos entradas a `self.processed_contours`. Cada entrada contiene el contorno y su tipo (ïnternal"), así como un ID único.

4. El contorno `container` también se agrega a `self.processed_contours`, y el ID del contorno `item` se vincula con el ID del contorno `container` en la clave `surface`.

5. La cadena `skin` se actualiza para incluir los nuevos identificadores de contornos internos.

6. Finalmente, se procesan los contornos de piel de `self.composition["skin"]`. Cada contorno de piel se agrega a `self.processed_contours` como un contorno .external", con el identificador `counter` y los identificadores de los contornos internos asociados.

7. La función termina al devolver la lista de contornos procesados.

6.9. Método `generate_gmsh_geo`

El método `generate_gmsh_geo` genera un archivo de geometría en formato GMSH (.geo) para la malla de un contorno de hueso. Este archivo es utilizado para la creación de mallas en simulaciones numéricas. El método escribe las coordenadas de los puntos, las líneas que los conectan, y las superficies de la geometría en el archivo de salida. Además, añade una caja que representa los límites del modelo y define entidades físicas para diferentes partes del modelo.

```
    def generate_gmsh_geo(self, output_file):
        with open(output_file, "w") as f:
            f.write("//Meshing bone.\n")
            f.write("//Inputs;\n")
            f.write("gridsize = 1;\n\n")

            point_index = 1
            line_index = 1
            surface_index = 1
            loop_index = 1
            max_perimeter = 0
            coords = ((0.0, 0.0), (0.0, 1.0), (1.0, 1.0),
               (1.0, 0.0), (0.0, 0.0))
            centroid = Polygon(coords)

            for contour in self.processed_contours:
                perimeter = Polygon(
```

```
                contour["item"] * self.pixel_spacing[0] *
                    1.0e-3
            ).length
            if max_perimeter < perimeter:
                max_perimeter = perimeter
                centroid = Polygon(
                    contour["item"] * self.pixel_spacing
                        [0] * 1.0e-3
                ).centroid
        for contour in self.processed_contours:
            point_index, line_index, loop_index = self.
                write_contour_to_geo(
                f, contour["item"], point_index,
                    line_index, loop_index, centroid
            )
            f.write(
                f'Plane Surface({surface_index}) = {{{
                    contour["surface"]}}};\n\n'
            )
            surface_index += 1

        point_index, line_index, loop_index = self.
            add_box_to_geo(
            f,
            point_index,
            line_index,
            loop_index,
        )
        self.add_physical_entities_to_geo(f,
            surface_index)
```

Descripción de los parámetros

- `output_file`: El archivo de salida donde se guardará la geometría en formato GMSH (.geo).
- `self.processed_contours`: Una lista de contornos procesados, que contiene la información de cada contorno, como los puntos y la superficie asociada.
- `self.pixel_spacing`: La resolución espacial de los píxeles en la imagen utilizada para generar los contornos.
- `gridsize`: El tamaño de la malla, que se establece en 1 para definir la resolución de la geometría.

Flujo de la función

1. La función comienza abriendo el archivo de salida `output_file` para escribir en él.

2. Escribe información preliminar, como los comentarios y el tamaño de la malla `gridsize`.

3. Inicializa los índices `point_index`, `line_index`, `surface_index` y `loop_index` para gestionar los puntos, líneas y superficies de la geometría.

4. Establece el contorno más grande (hueso) y calcula su perímetro y centroide, que se utilizará para centrar los contornos procesados.

5. Para cada contorno en `self.processed_contours`, se llama a la función `write_contour_to_geo` para escribir los puntos y líneas del contorno en el archivo. Luego, se genera una superficie asociada al contorno.

6. Se añade una caja delimitadora que representa los límites del modelo, usando la función `add_box_to_geo`.

7. Finalmente, se añaden las entidades físicas al archivo utilizando la función `add_physical_entities_to_geo`, que define las superficies físicas del modelo, como el "hueso", "músculo", çalcáneoz "trabecular".

Propósito de la función

El propósito de `generate_gmsh_geo` es crear un archivo de geometría en formato GMSH para la malla de un contorno de hueso, incluyendo las superficies correspondientes y las entidades físicas para su uso en simulaciones de malla. El archivo de salida permite realizar análisis numéricos de los contornos generados a partir de las imágenes procesadas.

6.10. Métodos de apoyo: `write_contour_to_geo`, `add_box_to_geo`, y `add_physical_entities_to_geo`

`write_contour_to_geo`

Este método escribe los puntos y líneas de un contorno en el archivo de geometría `GMSH`.

```
    def write_contour_to_geo(
        self, file, contour, point_index, line_index,
            loop_index, centroid
```

```
    ):
        centered_contour = contour * self.pixel_spacing[0] *
            1.0e-3 - np.array(
            [centroid.x, centroid.y]
        )

        # points = remove_duplicate_vectors(centered_contour
            [::10])  # Use every 10th point to reduce
            complexity
        points = (centered_contour[::7])
        for point in points:
            file.write(
                f"Point({point_index}) = {{{point[1]},{point
                    [0]},0,gridsize}};\n"
            )
            point_index += 1

        file.write("\n")

        lines = []
        for i in range(len(points)):
            next_i = (i + 1) % len(points)
            file.write(
                f"Line({line_index}) = {{{point_index-len(
                    points)+i},{point_index-len(points)+next_i
                    }}};\n"
            )
            lines.append(line_index)
            line_index += 1

        file.write("\n")

        file.write(f'Line Loop({loop_index}) = {{{",".join(
            map(str, lines))}}};\n')
        loop_index += 1

        return point_index, line_index, loop_index
```

Descripción:

- Este método toma un contorno, lo ajusta respecto a un centroide dado, y escribe los puntos del contorno en el archivo `GMSH`.
- Luego conecta estos puntos mediante líneas y define un `Line Loop` que representa el contorno cerrado.

`add_box_to_geo`

Este método añade una caja delimitadora al modelo, representando los límites del dominio de la malla.

```
    def add_box_to_geo(self, file, point_index, line_index,
        loop_index):
        L = self.box_size / 2
        box_points = [(-L, -L), (-L, L), (L, L), (L, -L)]

        for x, y in box_points:
            file.write(f"Point({point_index}) = {{{x},{y},0,
                gridsize}};\n")
            point_index += 1

        box_lines = []
        for i in range(4):
            next_i = (i + 1) % 4
            file.write(
                f"Line({line_index}) = {{{point_index-4+i},{
                    point_index-4+next_i}}};\n"
            )
            box_lines.append(line_index)
            line_index += 1

        file.write(f'Line Loop({loop_index}) = {{{",".join(
            map(str, box_lines))}}};\n')
        file.write(
            f"Plane Surface({loop_index+1}) = {{{loop_index
                },{loop_index-1}}};\n\n"
        )

        return point_index, line_index, loop_index + 2
```

Descripción:

- Define los puntos en las esquinas de una caja delimitadora basada en el tamaño del modelo (`box_size`).
- Conecta estos puntos mediante líneas, formando un contorno cerrado.
- Añade una superficie plana dentro de la caja, vinculándola a las líneas previamente definidas.

`add_physical_entities_to_geo`

Este método define entidades físicas en el archivo `GMSH`, categorizando las diferentes partes del modelo.

```
    def add_physical_entities_to_geo(self, file,
        surface_index):
        file.write(
            f'Physical Surface(100) = {{{surface_index+1}}};
                // "Medio de acoplamiento"\n'
        )
        file.write(f'Physical Surface(200) = {{{surface_index
            -1}}}; // "Musculo"\n')
        file.write(
            f'Physical Surface(300) = {{{surface_index-2},{
                surface_index-4}}}; // "Calcaneo"\n'
        )
        file.write(
            f'Physical Surface(400) = {{{surface_index-3},{
                surface_index-5}}}; // "Trabecular"\n'
        )
        file.write(f'Physical Line(10) = {{1,2,3,4}}; // "
            borde externo"\n')
        file.write("Mesh.CharacteristicLengthMax = 1.0e-2;\n"
            )
```

Descripción:

- Este método agrupa las superficies generadas en el modelo en entidades físicas, que se utilizan en simulaciones para identificar distintas regiones (e.g., hueso, músculo, etc.).
- También define las líneas físicas, como el borde externo del modelo.

Propósito General de los Métodos

- `write_contour_to_geo`: Define los contornos y líneas asociados a cada parte del modelo.
- `add_box_to_geo`: Añade límites al modelo, representados como una caja externa.
- `add_physical_entities_to_geo`: Clasifica las superficies y bordes en entidades físicas para facilitar su uso en análisis numéricos.

6.10.1. Resultados obtenidos

En la siguiente Figura 9 se mostrarán los resultados de la ejecución del comando

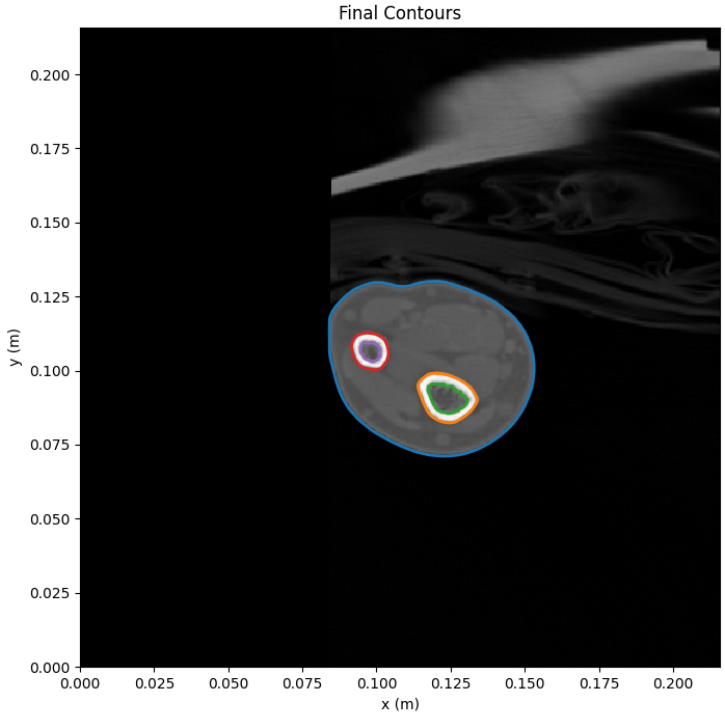

Figura 9: Se muestra la imagen procesada del Dicom, marcando los contornos de colores, los huesos trabeculares(color verde y violeta), los huesos corticales(color rojo y naranja)

```
gridsize = 1;

Point(1) = {0.004542358568493457,-0.005788357069207189 5,0,
    gridsize};
Point(2) = {0.006431681384799034 5,-0.007052821963207195,0,
    gridsize};
Point(3) = {0.007928880644751654,-0.006631333665207184,0,
    gridsize};
Point(4) = {0.009692649994241034,-0.00789579855920719,0,
    gridsize};
Point(5) = {0.011348644898449986,-0.009160263453207196,0,
    gridsize};
Point(6) = {0.012881121373243473,-0.010612409928958122,0,
    gridsize};
Point(7) = {0.01245963307524349,-0.012945500297181375,0,
    gridsize};
Point(8) = {0.010902909785932155,-0.014218123029207191,0,
```

```
    gridsize};
Point(9) = {0.009087726691243456,-0.015168968668297242,0,
    gridsize};
Point(10) = {0.006980285201243483,-0.016039103909488622,0,
    gridsize};
Point(11) = {0.004487482981643476,-0.015482587923207183,0,
    gridsize};
Point(12) = {0.0026600301467434723,-0.014639611327207189,0,
    gridsize};
Point(13) = {0.0012743151944147046,-0.012953658135207186,0,
    gridsize};
Point(14) = {0.00023647243324347034,-0.011184912598957195,0,
    gridsize};
Point(15) = {-0.0006025723689318946,-0.008738775155207185,0,
    gridsize};
Point(16) = {-0.000251006456867639,-0.006631333665207184,0,
    gridsize};
Point(17) = {0.0015009373272434762,-0.0056391576716850605,0,
    gridsize};
Point(18) = {0.0034917969475839056,-0.0057883570692071895,0,
    gridsize};

Line(1) = {1,2};
Line(2) = {2,3};
Line(3) = {3,4};
Line(4) = {4,5};
Line(5) = {5,6};
Line(6) = {6,7};
Line(7) = {7,8};
Line(8) = {8,9};
Line(9) = {9,10};
Line(10) = {10,11};
Line(11) = {11,12};
Line(12) = {12,13};
Line(13) = {13,14};
Line(14) = {14,15};
Line(15) = {15,16};
Line(16) = {16,17};
Line(17) = {17,18};
Line(18) = {18,1};

Line Loop(1) =
    {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
Plane Surface(1) = {1};
```

Todo esto representa un hueso trabecular.

6.10.2. Cierre del archivo

La parte final se divide en dos partes. La primera es crear un marco

```
Point(159) = {-0.125,-0.125,0,gridsize};
Point(160) = {-0.125,0.125,0,gridsize};
Point(161) = {0.125,0.125,0,gridsize};
Point(162) = {0.125,-0.125,0,gridsize};
Line(159) = {159,160};
Line(160) = {160,161};
Line(161) = {161,162};
Line(162) = {162,159};
Line Loop(6) = {159,160,161,162};
Plane Surface(7) = {6,5};
```

Se crean 4 puntos y cuatro líneas, generando un rectángulo. Al final se agregan las capas de hueso cortical, capas 6,5.

```
Physical Surface(100) = {7}; // "Medio de acoplamiento"
Physical Surface(200) = {5}; // "Musculo"
Physical Surface(300) = {4,2}; // "Calcaneo"
Physical Surface(400) = {3,1}; // "Trabecular"
Physical Line(10) = {1,2,3,4}; // "borde externo"
Mesh.CharacteristicLengthMax = 1.0e-2;
```

6.11. Conversión de archivos .geo a .msh

Para realizar la conversión de forma automática, se puede emplear el siguiente script en bash:

```
#!/bin/bash

# Carpeta donde est n los archivos .geo
input_folder="./"
output_folder="./msh"

# Crear carpeta de salida si no existe
mkdir -p "$output_folder"

# Iterar sobre cada archivo .geo
for geo_file in "$input_folder"/*.geo; do
    # Obtener el nombre base del archivo sin extensi n
    base_name=$(basename "$geo_file" .geo)
    # Convertir a .msh usando Gmsh
    gmsh "$geo_file" -2 -o "$output_folder/$base_name.msh"
done

echo "Conversi n completada. Archivos guardados en
   $output_folder"
```

En este script:

- `-2`: Genera una malla bidimensional (2D).
- `-o`: Indica el archivo de salida generado.
- `$input_folder` y `$output_folder`: Son las carpetas de entrada y salida correspondientes.

Cada archivo `.geo` en la carpeta de entrada es convertido en un archivo `.msh` y almacenado en la carpeta de salida especificada.

6.11.1. Modo de Uso

1. **Instalación requerida:** Gmsh(`https://gmsh.info/`)
2. **Ejecutar el script:** Ejecute el script con el siguiente comando:

```
./convert_geo_to_msh.sh
```

6.11.2. Ejemplo

A continuación se presentan dos figuras: una con una malla más fina (Fig. 10) y otra con una malla más gruesa (Fig. 11).

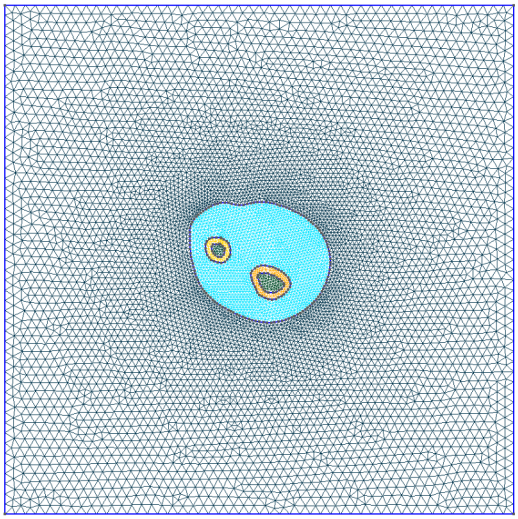

Figura 10: longitud máxima de los elementos de la malla es de 5.0e-3

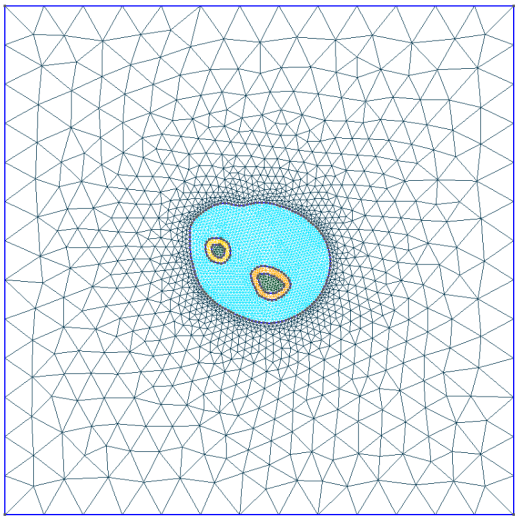

Figura 11: longitud máxima de los elementos de la malla es de 1.0e-2

6.12. Conversión FEM a FDTD

6.12.1. Diagrama de flujo general

El la Fig. 12 se muestra el diagrama de flujo que ilustra el funcionamiento general del script utilizado para la conversión de archivos FEM a FDTD

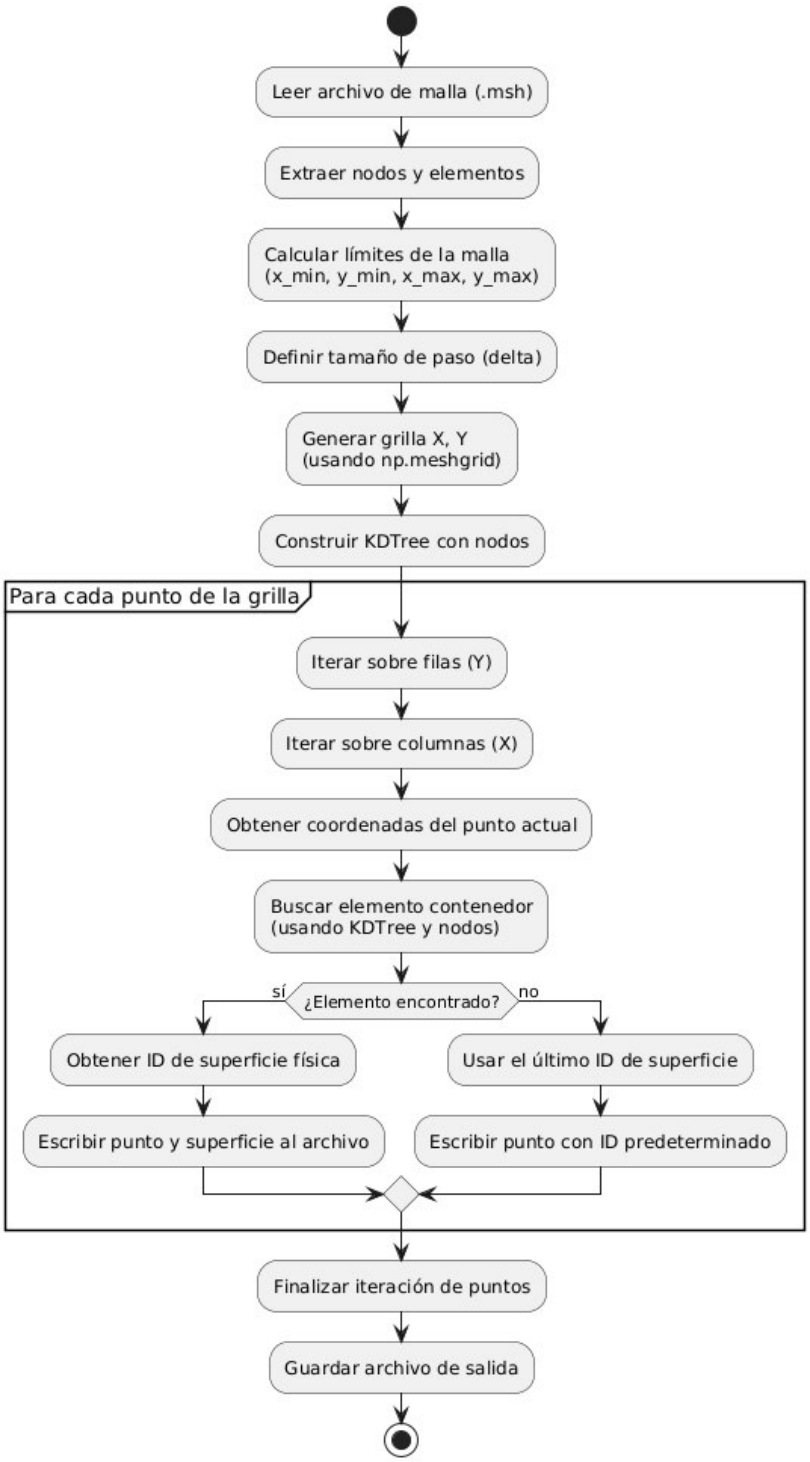

Figura 12: Diagrama de flujo del proceso de conversión de FEM a FDTD.

6.12.2. Script Principal

procesamiento se encuentra en el script `procimage2h5.py`. A continuación, se presenta un fragmento representativo del código:

```
import numpy as np
import meshio
from scipy.spatial import KDTree

mesh = meshio.read("munieca2dnewer.msh")

nodes = np.array(mesh.points)[:, :2]
print(nodes)
elements = mesh.cells_dict['triangle']

output_file = "output.5h"

physical_surfaces = mesh.cell_data_dict['gmsh:physical']['
    triangle']

x_min, y_min = np.min(nodes, axis=0)
x_max, y_max = np.max(nodes, axis=0)

delta = 0.02
dx = delta
dy = delta

print(f"Generando grilla con paso max{x_max} , y max {y_max}
    y paso {dx} y {dy}")

x_vals = np.arange(x_min, x_max + dx, dx)
y_vals = np.arange(y_min, y_max + dy, dy)
X, Y = np.meshgrid(x_vals, y_vals)

tree = KDTree(nodes)

def find_containing_element(point):
    distance, idx = tree.query(point, k=5)
    for elem in elements:
        if any(i in elem for i in idx):
            return elem
    return None

with open(output_file, 'w') as f:
    last_elem = 100
    for j, y in enumerate(Y):
        for i, x in enumerate(X):
            point = [x[i], y[j]]
            elem = find_containing_element(point)
```

```
            if elem is not None:
                surface_id = physical_surfaces[np.where((
                    elements == elem).all(axis=1))[0][0]]
                f.write(f"{i+1} {j+1} {surface_id}\n")
                last_elem = surface_id
            else:
                f.write(f"{i+1} {j+1} {last_elem}\n")

print(f"Grilla generada y guardada en {output_file}")
```

6.12.3. Modo de Uso

1. **Instalar dependencias:** Ejecute el siguiente comando para instalar los paquetes requeridos:

```
pip install -r requirements.txt
```

2. **Ejecutar el script:** Ejecute el script con el siguiente comando:

```
python3 procimage2h5.py
```

6.12.4. Código para Generación de Grilla y Asignación de Elementos

Lectura de la malla El siguiente bloque de código utiliza `meshio` para leer un archivo de malla 2D en formato `.msh`. Se extraen los nodos (coordenadas de los puntos) y los elementos triangulares:

```
import numpy as np
import meshio
from scipy.spatial import KDTree

mesh = meshio.read("munieca2dnewer.msh")

nodes = np.array(mesh.points)[:, :2]
print(nodes)
elements = mesh.cells_dict['triangle']

output_file = "output.5h"

physical_surfaces = mesh.cell_data_dict['gmsh:physical']['
    triangle']
```

Definición de la grilla regular Se calcula una grilla rectangular basada en las dimensiones de los nodos extraídos de la malla:

```
x_min, y_min = np.min(nodes, axis=0)
x_max, y_max = np.max(nodes, axis=0)

delta = 0.02
dx = delta
dy = delta

print(f"Generando grilla con paso max{x_max} , y max {y_max}
    y paso {dx} y {dy}")

x_vals = np.arange(x_min, x_max + dx, dx)
y_vals = np.arange(y_min, y_max + dy, dy)
X, Y = np.meshgrid(x_vals, y_vals)
```

Descripción:

- x_min,y_min,x_max,y_max: Son los límites de nodos
- delta: Es la resolución en grilla, entre más grande el número menor es la resolución y mayor es el tiempo de ejecución
- x_vals, y_vals : Se genera un array que comienza en x_min o y_min, incrementándose de a dx en cada paso, y termina antes de alcanzar x_max + dx o y_max. Ejemplo de array generado: [0. 0.2 0.4 0.6 0.8 1.]
- X, Y: La función np.meshgrid(x_vals, y_vals) genera dos matrices 2D que representan las coordenadas de una grilla cartesiana a partir de dos vectores unidimensionales. La matriz X contiene las coordenadas x repetidas en cada fila, mientras que la matriz Y contiene las coordenadas y repetidas en cada columna. En el código, esto se utiliza para construir una grilla de puntos basada en los rangos de x e y, necesarios para evaluar cada punto contra los elementos de la malla.

Búsqueda de Elementos Contenedores con KDTree Utilizando `KDTree` de `scipy.spatial`, se busca encontrar el elemento triangular que contiene cada punto de la grilla:

```
tree = KDTree(nodes)

def find_containing_element(point):
    distance, idx = tree.query(point, k=5)
    for elem in elements:
```

```
        if any(i in elem for i in idx):
            return elem
    return None
```

Creación del KDTree El `KDTree` es una estructura de datos proporcionada por `scipy.spatial` que permite realizar búsquedas rápidas de los vecinos más cercanos en un espacio multidimensional. En este caso, se crea a partir de las coordenadas de los nodos de la malla:

```
tree = KDTree(nodes)
```

El objeto `tree` permite consultar los nodos más cercanos a un punto dado de manera eficiente.

Función `find_containing_element` Esta función identifica el elemento triangular que contiene un punto específico, utilizando el `KDTree` para optimizar la búsqueda. Su definición es la siguiente:

```
def find_containing_element(point):
    distance, idx = tree.query(point, k=5)
    for elem in elements:
        if any(i in elem for i in idx):
            return elem
    return None
```

Entradas y Salidas: - `point`: Un punto de la grilla (coordenadas x, y) que se desea analizar. - Devuelve el elemento triangular (`elem`) que contiene al punto o `None` si no se encuentra ninguno.

Proceso: 1. Búsqueda de nodos cercanos:

```
distance, idx = tree.query(point, k=5)
```

- Encuentra los `k=5` nodos más cercanos al punto dado `point`. - `idx`: Contiene los índices de los nodos más cercanos. - `distance`: Distancias de los nodos al punto (no se utiliza en este caso).

2. Verificación del elemento contenedor:

```
for elem in elements:
    if any(i in elem for i in idx):
        return elem
```

- Se itera sobre cada triángulo `elem` en `elements`. - Si alguno de los nodos cercanos pertenece al triángulo `elem`, este se devuelve como el contenedor del punto.

3. Caso por defecto:

```
return None
```

- Si ningún triángulo contiene a los nodos cercanos, la función retorna `None`.

Propósito Esta función optimiza la búsqueda de elementos contenedores utilizando el `KDTree` para localizar rápidamente los nodos más cercanos, simplificando la identificación del triángulo contenedor en la malla.

Escritura del archivo de salida Se genera un archivo de salida que asigna el ID de la superficie física a cada punto de la grilla:

```
output_file = "output.5h"

with open(output_file, 'w') as f:
    last_elem = 100  # Valor por defecto para elementos no encontrados
    for j, y in enumerate(Y):
        for i, x in enumerate(X):
            point = [x[i], y[j]]  # Punto de la grilla
            elem = find_containing_element(point)

            if elem is not None:
                # Encontrar el ID de la superficie
                surface_id = physical_surfaces[np.where((elements == elem).all(a
                f.write(f"{i+1} {j+1} {surface_id}\n")
                last_elem = surface_id
            else:
                # Usar el último ID válido
                f.write(f"{i+1} {j+1} {last_elem}\n")

print(f"Grilla generada y guardada en {output_file}")
```

Resumen Este código permite generar una grilla regular basada en un conjunto de nodos extraídos de una malla y asigna a cada punto el ID de la superficie física correspondiente. Si un punto no está contenido en ningún elemento, se le asigna el último ID encontrado.

6.13. Generación de una Imagen a partir de Datos de un Archivo

El siguiente código en Python utiliza la librería `PIL` para crear una imagen a partir de un archivo que contiene datos sobre las coordenadas y los colores de las celdas en la imagen. A continuación, se detallan las funciones y pasos principales del código.

6.13.1. Diagrama de flujo general

El la Fig. 13 se muestra el diagrama de flujo que ilustra el funcionamiento general del script utilizado para la conversión de archivos a una imagen.

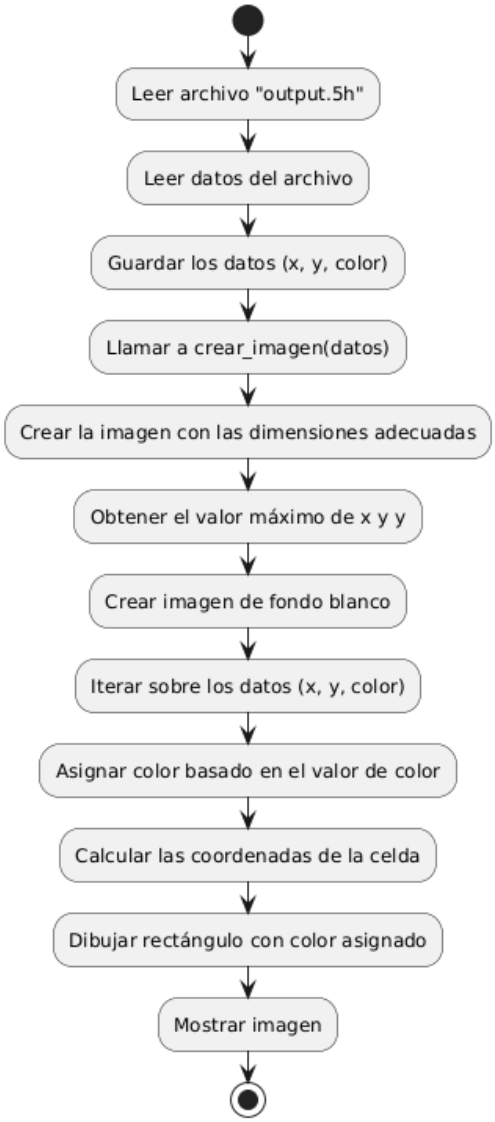

Figura 13: Diagrama de flujo del proceso de conversión del archivo a png

6.13.2. Script Principal

procesamiento se encuentra en el script `convertToPng.py`. A continuación, se presenta un fragmento representativo del código:

```
from PIL import Image, ImageDraw

# Leer el archivo y cargar los datos
def leer_datos(archivo):
    datos = []
```

```
    with open(archivo, 'r') as f:
        for linea in f:
            # Dividir la l nea en tres columnas (x, y, color
               )
            x, y, color = map(int, linea.split())
            datos.append((x, y, color))
    return datos

# Crear la imagen a partir de los datos
def crear_imagen(datos, ancho_celda=20, alto_celda=20):
    # Obtener las dimensiones m ximas del eje x e y
    max_x = max(d[0] for d in datos)
    max_y = max(d[1] for d in datos)

    # Crear una imagen con el tama o adecuado
    ancho = max_x * ancho_celda
    alto = max_y * alto_celda
    imagen = Image.new("RGB", (ancho, alto), "white")
    draw = ImageDraw.Draw(imagen)

    # Dibujar cada celda basada en los datos
    for x, y, color in datos:
        # Determinar el color basado en el valor de la
           tercera columna
        color_celda = (255, 255, 255) if color == 0 else (0,
           0, 0)  # blanco para 0, negro para 100
        color_celda = (255, 0, 255) if color == 200 else
           color_celda
        color_celda = (255, 0, 0) if color == 300 else
           color_celda
        color_celda = (255, 255, 0) if color == 400 else
           color_celda
        # Calcular las coordenadas de la celda
        esquina_superior_izq = ((x - 1) * ancho_celda, (y -
           1) * alto_celda)
        esquina_inferior_der = (x * ancho_celda, y *
           alto_celda)

        # Dibujar el rect ngulo (cuadrado)
        draw.rectangle([esquina_superior_izq,
           esquina_inferior_der], fill=color_celda)

    return imagen

# Ruta del archivo con los datos
ruta_archivo = "output.5h"

# Leer los datos del archivo
datos = leer_datos(ruta_archivo)
```

```
# Crear la imagen
imagen = crear_imagen(datos)

# Guardar o mostrar la imagen
imagen.show()  # Muestra la imagen
# imagen.save("imagen_resultante.png")  # O para guardar la
    imagen
```

6.13.3. Modo de Uso

1. **Instalar dependencias:** Ejecute el siguiente comando para instalar los paquetes requeridos:

```
pip install -r requirements.txt
```

2. **Ejecutar el script:** Ejecute el script con el siguiente comando:

```
python3 convertToPng.py
```

6.13.4. Código para Generación de Grilla y Asignación de Elementos

- **leer_datos(archivo)**: Esta función lee el archivo que contiene los datos. Cada línea del archivo tiene tres valores: las coordenadas `x` y `y` de una celda, y un valor de color. La función devuelve una lista de tuplas `(x, y, color)` que se utilizarán para dibujar la imagen.
- **crear_imagen(datos, ancho_celda=20, alto_celda=20)**: Esta función crea la imagen a partir de los datos proporcionados. Primero, obtiene las dimensiones máximas de las coordenadas `x` e `y` para determinar el tamaño adecuado de la imagen. Luego, crea una imagen en blanco y dibuja un rectángulo para cada celda, utilizando el valor de `color` para determinar el color de la celda. Los colores se asignan de la siguiente manera:
  - `0` → Blanco: `(255, 255, 255)`
  - `100` → Negro: `(0, 0, 0)`
  - `200` → Magenta: `(255, 0, 255)`
  - `300` → Rojo: `(255, 0, 0)`
  - `400` → Amarillo: `(255, 255, 0)`

Cada celda se dibuja con un tamaño de `ancho_celda` y `alto_celda`, y las coordenadas `(x, y)` se usan para determinar su posición.

- **Proceso de ejecución**: El archivo `output.5h` es leído utilizando la función `leer_datos`. Luego, se pasa la lista de datos a `crear_imagen`, que genera la imagen. Finalmente, se muestra la imagen utilizando `imagen.show()`, o alternativamente, se puede guardar con `imagen.save()`.

6.13.5. Resultados

Entre más bajo sea el delta, más resolución tendrá a costa de que el proceso demore más tiempo. Esto se debe a que la búsqueda es más lenta, al tener que buscar más referencias cercanas y realizar más búsquedas.Como se muestra en las siguientes Figura 14 y Figura 15.

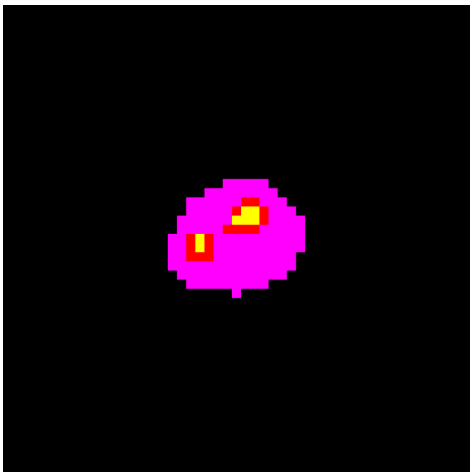

Figura 14: Con un delta de 0.005, una resolución de 50X50, y aproximadamente 2 minutos de procesamiento.

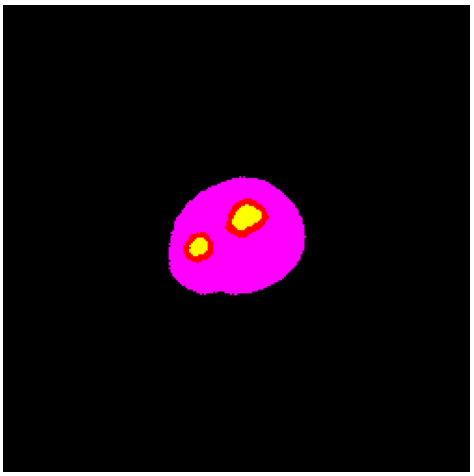

Figura 15: Con un delta de 0.001, una resolución de 250X250, y aproximadamente 1 hora de procesamiento.

7. Conclusiones

7.1. Resumen de los resultados

Se desarrolló una herramienta en Python que implementa el algoritmo *marching squares* para segmentación y filtra automáticamente las imágenes Dicom provenientes de tomografía convencional. A través de diversas bibliotecas especializadas, el algoritmo identifica formas y, utilizando un umbral preestablecido, clasifica distintos tipos de tejidos, como huesos cortical, trabecular y el resto de los tejidos blandos (músculos, tendones, etc.). Posteriormente, se aplican filtros lógicos que descartan las imágenes que no cumplan con ciertos requisitos. Luego, la herramienta de software desarrollada escribe la malla en un formato CAD preestablecido de Elementos Finitos (FEM).

En un caso práctico, esta herramienta procesó 237 archivos, de los cuales 39 cumplieron los requisitos para ser utilizados. El tiempo total de procesamiento fue de 3.76 segundos.

Adicionalmente, se desarrolló otra herramienta poco optimizada que convierte la malla FEM en FDTD. Sin embargo, este algoritmo aún presenta problemas de eficiencia: el tiempo de procesamiento varía significativamente según la resolución, desde aproximadamente 1 minuto para resoluciones moderadas hasta cerca de 1 hora para resoluciones más altas, mostrando un comportamiento exponencial en el tiempo requerido.

7.2. Posibles mejoras

Lo dividiremos en dos partes, la primera propondré posibles mejoras al script de conversión de Dicom a .geo. En la segunda propondré mejoras para el script que convierte FEM a FDTD.

7.2.1. Posibles Mejoras para el Script conversión de Dicom a .geo

El manejo de errores y validación es fundamental para garantizar un proceso robusto. Se debe agregar validaciones que verifiquen la validez de los archivos Dicom y asegurar el manejo adecuado de excepciones para archivos corruptos. Además, es importante validar los parámetros de entrada, como los umbrales de contorno, para evitar errores durante el procesamiento.

En cuanto a la configurabilidad, es necesario permitir que los umbrales de contorno sean configurables a través de la línea de comandos. Parámetros como `box_size` deben ser personalizables, y se deben agregar opciones para ajustar la reducción de puntos en los contornos generados.

Para optimizar el rendimiento, se propone implementar procesamiento paralelo de archivos Dicom, optimizar los algoritmos de detección y filtrado de contornos, y considerar bibliotecas de procesamiento de imágenes más eficientes.

El registro y depuración también son áreas clave. Se debe incluir un registro detallado del proceso de conversión, implementar niveles de verbosidad para ajustar la cantidad de información mostrada, y agregar mensajes de depuración explicativos que faciliten la identificación de problemas.

En cuanto a la visualización, es necesario mejorar las funciones actuales, permitir la generación y almacenamiento de imágenes de los contornos procesados, y agregar opciones para representar visualmente los resultados de manera clara y comprensible.

Finalmente, para aumentar la flexibilidad en los tipos de imagen, se debe extender el soporte para diferentes modalidades de imágenes médicas, implementar preprocesamiento adaptativo y manejar diversos espaciados de píxeles y grosores de corte.

7.2.2. Posibles Mejoras para el Script de Procesamiento de Malla

El manejo de errores y la validación también son esenciales en el procesamiento de mallas. Se recomienda agregar validaciones para los archivos de malla de entrada, implementar manejo de excepciones para archivos inválidos y verificar la integridad de los datos de la malla antes del procesamiento.

En cuanto a configurabilidad, es importante permitir que el tamaño de paso de la cuadrícula sea configurable mediante argumentos, hacer que el método de búsqueda de elementos contenedores sea ajustable, y agregar opciones para personalizar la estrategia de interpolación utilizada.

Para mejorar el rendimiento, se deben optimizar métodos como `find_containing_element()`, considerar el uso de estructuras de datos más eficientes para la búsqueda espacial, implementar procesamiento paralelo para conjuntos de datos grandes, y reducir al máximo área a analizar para disminuir el tiempo de ejecución.

En el área de registro y depuración, se recomienda incluir un registro detallado del proceso de generación de cuadrículas, implementar niveles de verbosidad ajustables, y agregar métricas de rendimiento y estadísticas para el análisis posterior.

La visualización y análisis también pueden beneficiarse de nuevas funciones para visualizar la cuadrícula generada, opciones para generar informes de procesamiento, y la capacidad de exportar resultados en diferentes formatos.

En cuanto a la flexibilidad de entrada y salida, se debe asegurar el soporte para múltiples formatos de archivos de malla, permitir configuraciones

flexibles de archivos de salida, y manejar diferentes sistemas de coordenadas y unidades.

Para mejorar los algoritmos avanzados, se pueden implementar métodos de interpolación más sofisticados, estrategias de manejo para elementos parcialmente contenidos, y considerar técnicas de suavizado o refinamiento de malla.

Por último, en términos de seguridad y robustez, se recomienda agregar validaciones para prevenir desbordamientos, manejar casos extremos de geometría, y añadir opciones de recuperación ante fallos para garantizar un funcionamiento estable del sistema.

7.3. Extensiones del proyecto

Existen varias posibilidades para extender este proyecto y aumentar su alcance y utilidad. Una opción es desarrollar un paquete que pueda ser utilizado por la comunidad, facilitando su distribución y accesibilidad. Además, se propone trasladar el proyecto a un contenedor Docker, lo que permitiría su ejecución en cualquier entorno de manera sencilla y sin problemas de configuración.

Otra línea de desarrollo consiste en modificar el enfoque del script que procesa las mallas, tratando estas como si fueran imágenes. Esto permitiría, a partir de una representación visual, realizar la conversión de métodos basados en el modelo de FEM al modelo de FDTD.

Finalmente, se podría implementar un nuevo script que realice la conversión inversa, es decir, de FDTD a FEM, completando así la interoperabilidad entre ambos modelos y ampliando las aplicaciones prácticas del proyecto.

8. Referencias

[1] J. M. Kling, B. L. Clarke y N. P. Sandhu. «Osteoporosis prevention, screening, and treatment: a review». En: *Journal of Women's Health* 23.7 (2014), págs. 563-572.

[2] M. Pastorino. *Microwave Imaging*. John Wiley & Sons, 2010.

[3] X. Chen. *Computational Methods for Electromagnetic Inverse Scattering*. John Wiley & Sons, 2018.

[4] Mikael Persson, Andreas Fhager, Hana Dobšíček Trefná, Yinan Yu, Tomas McKelvey, Göran Pegenius, Jan-Erik Karlsson y Mikael Elam. «Microwave-based stroke diagnosis making global prehospital thrombolytic treatment possible». En: *IEEE Transactions on Biomedical Engineering* 61.11 (2014), págs. 2806-2817.

[5] Christophe Geuzaine y Jean-François Remacle. «Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities». En: *International journal for numerical methods in engineering* 79.11 (2009), págs. 1309-1331.

[6] A. Zakaria, C. Gilmore y J. LoVetri. «Finite-element contrast source inversion method for microwave imaging». En: *Inverse Problems* 26.11 (2010), pág. 115010.

[7] N. AlSawaftah, S. El-Abed, S. Dhou y A. Zakaria. «Microwave imaging for early breast cancer detection: Current state, challenges, and future directions». En: *Journal of Imaging* 8.5 (2022), pág. 123.

[8] A. Fhager, S. Candefjord, M. Elam y M. Persson. «Microwave diagnostics ahead: Saving time and the lives of trauma and stroke patients». En: *IEEE Microwave Magazine* 19.3 (2018), págs. 78-90.

[9] R. M. Irastorza, E. Blangino, C. M. Carlevaro y F. Vericat. «Modeling of the dielectric properties of trabecular bone samples at microwave frequency». En: *Medical & Biological Engineering & Computing* 52 (2014), págs. 439-447.

[10] Daniel SH Lo. *Finite element mesh generation*. CRC press, 2014.

[11] William E Lorensen y Harvey E Cline. «Marching cubes: A high resolution 3D surface construction algorithm». En: *Seminal graphics: pioneering efforts that shaped the field*. 1998, págs. 347-353.