



Integración de Biblioteca de Procesamiento de Fibras Cerebrales

Gonzalo Sabat Arriagada

Profesora Patrocinante: Cecilia Hernández

Profesora Co-Guía: Pamela Guevara

17 de Noviembre de 2021

RESUMEN

El cerebro es la parte más importante del cuerpo y por lo tanto, estudiar el cerebro es un área de sumo interés. Por esto mismo, el Grupo de Imágenes Médicas de la Universidad de Concepción ha desarrollado múltiples algoritmos para el análisis y estudio de las fibras neuronales que lo componen, sin embargo, la falta de centralización y documentación de las implementaciones complica el uso de estas. En esta memoria, se desarrolló una biblioteca en Python que incluye implementaciones en distintos lenguajes de programación, casos de prueba, documentación y manual de usuario. Los resultados obtenidos presentan una solución eficaz a los problemas encontrados, mediante la utilización de la plataforma PyPI, como repositorio de paquetes y bibliotecas.

Índice

| | |
|--|-----------|
| 1. Introducción | 4 |
| 1.1. Objetivo General | 7 |
| 1.2. Objetivos Específicos | 7 |
| 1.3. Efectos Esperados | 7 |
| 2. Discusión Bibliográfica | 8 |
| 2.1. Segmentación | 8 |
| 2.2. Clustering Jerárquico | 8 |
| 2.3. FFClust | 9 |
| 2.4. DIPY - Diffusion Imaging in Python | 9 |
| 2.5. PyPI - Python Packaging Index | 10 |
| 2.6. Doxygen | 10 |
| 2.7. Problema Encontrado | 10 |
| 3. Modelo Desarrollado | 13 |
| 3.1. Desarrollo del Paquete | 14 |
| 3.1.1. Preparación | 14 |
| 3.1.2. Archivo <i>setup.py</i> | 16 |
| 3.1.3. Archivo <i>MANIFEST.in</i> | 17 |
| 3.1.4. Modularización de los códigos | 20 |
| 3.1.5. Módulo <i>segmentation</i> | 30 |
| 3.1.6. Módulo <i>ffclust</i> | 31 |
| 3.1.7. Módulo <i>hclust</i> | 32 |
| 3.1.8. Sub-módulo <i>utils.intersection</i> | 33 |
| 3.1.9. Sub-módulo <i>utils.deform</i> | 34 |
| 3.1.10. Sub-módulo <i>utils.sampling</i> | 34 |
| 3.1.11. Sub-módulo <i>utils.postprocessing</i> | 35 |
| 3.2. Documentación | 36 |
| 3.3. Subida a PyPI | 38 |
| 4. Experimentos y Resultados | 40 |
| 4.1. <i>Segmentation</i> | 41 |
| 4.2. <i>FFclust</i> | 42 |
| 4.3. <i>Hclust</i> | 43 |
| 4.4. <i>Utils</i> | 43 |
| 4.5. Documentación | 44 |
| 4.6. Instalación en Windows | 45 |

| | |
|--------------------------------|-----------|
| 4.7. Resultado Final | 46 |
| 5. Conclusiones | 47 |

Índice de figuras

| | |
|--|----|
| 1.1. (a) Imagen de Resonancia Magnética por difusión y (b) Representación 3D de las fibras que conforman un tractograma o tractografía cerebral de cerebro completo. | 4 |
| 3.1. Estructura jerárquica de la biblioteca <i>Phybers</i> | 14 |
| 3.2. Estructura de la última versión local del paquete <i>phybers</i> | 16 |
| 3.3. Contenido del archivo <i>setup.py</i> de la biblioteca. | 17 |
| 3.4. Extracto del contenido del archivo <i>MANIFEST</i> | 19 |
| 3.5. Estructura del paquete local <i>example</i> | 20 |
| 3.6. Contenido del archivo <i>__main__.py</i> del paquete <i>example</i> | 21 |
| 3.7. Ejecución del archivo <i>__main__.py</i> | 21 |
| 3.8. Estructura jerárquica del módulo <i>segmentation</i> | 22 |
| 3.9. Demostración gráfica de entrada y salida del módulo <i>segmentation</i> | 30 |
| 3.10. Representación gráfica de entrada y salida del módulo <i>ffclust</i> . Aquellos notados con * son argumentos opcionales. | 31 |
| 3.11. Representación gráfica de entrada y salida del módulo <i>hclust</i> | 32 |
| 3.12. Representación gráfica de entrada y salida del módulo <i>intersection</i> | 33 |
| 3.13. Representación gráfica de entrada y salida del módulo <i>deform</i> | 34 |
| 3.14. Representación gráfica de entrada y salida del módulo <i>sampling</i> | 34 |
| 3.15. Representación gráfica de entrada y salida del módulo <i>postprocessing</i> | 35 |
| 3.16. Comentarios con notación de Doxygen en código de módulo <i>segmentation</i> | 36 |
| 3.17. Documentación generada por Doxygen, en formato HTML. | 37 |
| 3.18. Versión 0.0.45 del paquete <i>segtest</i> en el sitio de PyPI. | 38 |
| 4.19. Página de inicio de la documentación generada por Doxygen. | 45 |

1. Introducción

El cerebro es el órgano principal del cerebro humano, se encarga de mantener funciones vitales del cuerpo, tanto fisiológicas como psicológicas. Así mismo, por las mismas razones este es susceptible a trastornos psiquiátricos, anomalías y enfermedades degenerativas que afectan múltiples funcionalidades del ser humano.

Es precisamente por esto que el estudio del cerebro humano es un área de estudio con un interés cada vez mayor. La posibilidad de encontrar nuevos tratamientos y curas para enfermedades que en la actualidad son sumamente complejos sería un avance científico que podría salvar vidas.

Para poder estudiar el cerebro, se utiliza dMRI (observar figura 1a). Esta es una técnica que permite obtener imágenes de la estructura del tejido cerebral e inferir su conectividad mediante la difusión de las moléculas de agua en el cerebro. Esto ha mejorado, por ejemplo, considerablemente el manejo de derrames cerebrales [1,2]. En conjunto con técnicas de dMRI, se utilizan algoritmos de tractografía, una técnica de modelado que permite reconstruir tractos nerviosos utilizando la información obtenida por dMRI. Esto permite generar tractogramas (observar figura 1b), que son conjuntos de polilíneas 3D, formadas por puntos 3D, que representan las trayectorias de los principales fascículos cerebrales [3].

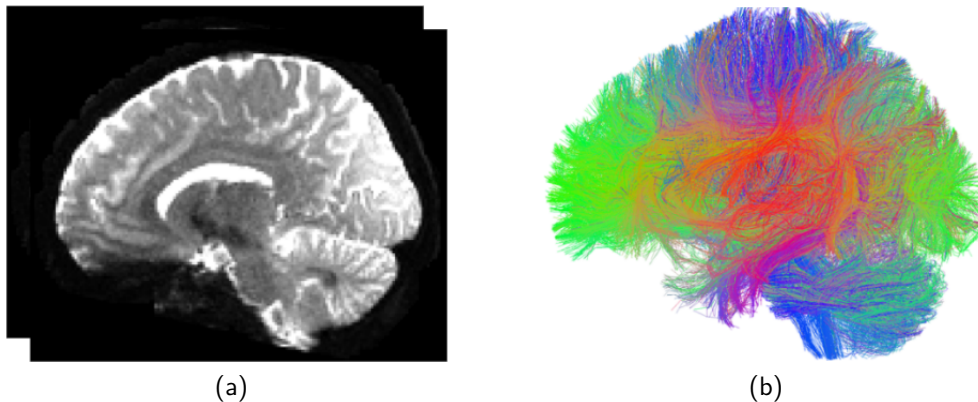


Figura 1.1: (a) Imagen de Resonancia Magnética por difusión y (b) Representación 3D de las fibras que conforman un tractograma o tractografía cerebral de cerebro completo.

El estudio del conectoma humano tiene el propósito de describir las conexiones cerebrales con un gran nivel de detalles, así como las regiones cerebrales que son conectadas y las funciones asociadas a estas regiones. Las 2 principales estrategias para el estudio de las conexiones cerebrales dadas por tractogramas, son:

1. Segmentación de fibras cerebrales [4], donde se realiza una identificación de fascículos de fibras (o grupos de fibras) conocidos a partir de un atlas de fascículos, basado en una distancia entre los puntos de las fibras. Los atlas de fibras están formados por los fascículos conocidos y previamente estudiados de múltiples sujetos. Los algoritmos de segmentación comparan las fibras de la tractografía de un sujeto, con las fibras del atlas, creando agrupaciones en base a los fascículos de dicho atlas.
2. Clustering de fibras cerebrales [3,5], que realiza un agrupamiento de fibras usado para grupos de fibras similares, considerando su forma y posición. Como medida de similitud se utiliza la distancia entre pares de fibras que se calcula a partir de las coordenadas de los puntos de las fibras cerebrales en el espacio euclidiano [3]. Los algoritmos de agrupamiento de fibras cerebrales generalmente aplican técnicas del aprendizaje no supervisado. A diferencia de la segmentación de fibras, el clustering de fibras permite descubrir nuevas asociaciones de fibras. Esto posibilita estudiar la estructura intrínseca de los conjuntos de tractografía, debido a que los algoritmos de agrupamientos entregan todos los grupos de fibras encontrados en un conjunto de tractografía cerebral.

El Grupo de Análisis de Imágenes Médicas, dirigido por la Facultad de Ingeniería de la Universidad de Concepción ha desarrollado varios algoritmos para el análisis de fibras cerebrales. Entre ellos se destacan, la segmentación de fibras cerebrales mediante un atlas multisujeto [4, 6, 7], el clustering jerárquico [8] y FFClust (Fast Fiber Clustering,[5]). Además de un conjunto de herramientas útiles que permiten pre-procesar los tractogramas, extraer medidas estadísticas para evaluar el desempeño de los algoritmos de segmentación y de clustering de fibras cerebrales, entre otras herramientas. Estas implementaciones han sido desarrollados por diferentes alumnos tesis de pregrado y postgrado en distintos lenguajes de programación. Es por ello que son códigos complicados de usar, ya que cada uno debe instalarse y aprender a usarse de forma separada. No solo eso, sino que no existe documentación que facilite su utilización. Aunque estos algoritmos están disponibles para la comunidad científica, todo lo anterior constituye una limitación para la utilización de estos. En la actualidad hay diversas herramientas

desarrolladas para el estudio del conectoma humano, como por ejemplo la biblioteca DIPY [9] desarrollada en Python. Sin embargo, DIPY no tiene disponible el algoritmo de segmentación de fibras cerebrales mediante atlas multisujeto [4] y el algoritmo de clustering FFClust demostró obtener clústeres más compactos en un menor tiempo de cálculo que su equivalente Quickbundles (implementado en DIPY). Además, esta no cuenta con un paquete de herramientas para evaluar el resultado del clustering o segmentación de fibras cerebrales.

El desarrollo de esta biblioteca permitirá integrar y estructurar la implementación del algoritmo de segmentación, clustering y herramientas para el análisis de fibras cerebrales desarrolladas por el Grupo de Análisis de Imágenes Médicas de nuestra Universidad. Esto entregará un grupo de algoritmos y herramientas de fácil acceso y uso que permitirán contribuir al estudio del conectoma humano.

1.1. Objetivo General

Unificar y documentar implementaciones de software existentes sobre procesamiento de fibras cerebrales y adaptarlos en una biblioteca para facilitar su uso y aplicación en el procesamiento de fibras cerebrales o clusters.

1.2. Objetivos Específicos

- Estudiar los algoritmos y los formatos de datos presentes en los códigos existentes.
- Crear una biblioteca de Python que integre los cuatro módulos para procesar las fibras cerebrales.
- Generar documentación y datos de ejemplo para el uso de las funcionalidades e instrucciones de instalación de la biblioteca.
- Definir y llevar a cabo las pruebas de software necesarias que permitan verificar la funcionalidad proporcionada en los diferentes módulos de la biblioteca.
- Definir repositorio caracterizado para datos de prueba.

1.3. Efectos Esperados

- Apoyar la investigación llevada a cabo por el Grupo de Análisis de Imágenes Médicas.
- Mejorar la accesibilidad de los algoritmos para estudiantes de pregrado y postgrado.
- Permitir la asequibilidad a académicos externos a la Universidad a los algoritmos desarrollados por el Grupo de Análisis de Imágenes Médicas

2. Discusión Bibliográfica

En esta sección se presentan los distintos algoritmos cuyos códigos guardan relación con el trabajo realizado y conceptos fundamentales para su comprensión. Además, se mencionan brevemente la biblioteca que inspiró la realización de este proyecto de memoria de título, el repositorio de paquetes de Python al que se subió el paquete, la herramienta para generar documentación utilizada y las razones por las que se escogieron estas. Finalmente, se presenta el problema que el trabajo presente propone resolver.

2.1. Segmentación

La Segmentación de fibras cerebrales basado en atlas multisujeto [4, 6 y 7], tiene como objetivo clasificar las fibras de los sujetos en función de un atlas de fascículos multisujeto [4]. Para clasificar las fibras de un sujeto se utiliza la máxima distancia Euclidiana entre cada fibra del sujeto y cada centroide del atlas, manteniendo solamente las fibras con una distancia por debajo de un umbral definido para cada fascículo del atlas. Luego en [6], se implementó en el lenguaje de programación C y se explotó el paralelismo a nivel de subprocesos en múltiples CPUs para tratar con conjuntos de datos de tractografía masiva y lograr un menor tiempo de ejecución. Por último, en [7] fue mejorado el algoritmo [6] y su implementación, con el objetivo de reducir el uso de recursos informáticos, principalmente CPU y memoria. La nueva implementación utiliza el lenguaje de programación C++, para mejora la modularidad y la extensibilidad de la versión anterior. Las optimizaciones se centraron en reducir la memoria utilizada para resultados temporales, mejorando la paralelización de tareas así como el algoritmo de descarte rápido de fascículos, y la clasificación de las fibras utilizando solo el fascículo más cercano. El algoritmo optimizado mejora tanto el tiempo de ejecución como el uso de la memoria de su predecesor.

2.2. Clustering Jerárquico

Clustering Jerárquico [8] es un algoritmo de agrupamiento basado en la distancia euclidiana entre pares de fibras. Originalmente aplicado entre sujetos, como parte de un método automático para identificar fascículos de fibras cortas reproducibles en una base de datos HARDI. Este algoritmo calcula una matriz distancia para el conjunto de fibras cerebrales. Luego, a partir de la matriz, se calcula un grafo de afinidad. La afinidad se define como

$$a_{ij} = e^{\frac{-d_{ij}}{\sigma^2}}$$

donde d_{ij} es la distancia entre los elementos i y j , y σ^2 es un parámetro que define la escala de similitud (60 mm). El cálculo de la afinidad se realiza para cada par de fibras con una distancia menor que una distancia máxima definida. A partir del grafo de afinidad se genera el árbol jerárquico mediante un algoritmo de clustering jerárquico aglomerativo average-link. El árbol se particiona utilizando un umbral de distancia. El algoritmo clustering jerárquico se implementó en Python 2.7 y en lenguaje C++.

2.3. FFClust

FFClust [5], es un algoritmo de agrupamiento intra-sujeto basado en K-Means por puntos de las fibras. Este se puede aplicar a las fibras de la tractografía de todo el cerebro. Tiene como objetivo crear grupos similares de fibras de acuerdo a su posición. La similitud entre fibras se define por el máximo de la distancia Euclidiana entre puntos correspondientes. Esta se considera una distancia restrictiva, ya que cualquier diferencia local relevante entre las fibras, en las extremidades de la fibra y en toda su forma, se capturará exitosamente. Se utilizó el método Elbow [11] para determinar el número óptimo de grupos de K-Means. FFClust se implementó en Python versión 3.6 y en lenguaje C usando el compilador g++ versión 7.4.0. El algoritmo admite la ejecución secuencial y paralela mediante OpenMP.

2.4. DIPY - Diffusion Imaging in Python

DIPY [9] es una biblioteca de Python que contiene diversos tipos de metodología para la visualización y el análisis estadístico de imágenes médicas. DIPY posee muchas características que la hacen ser una de las bibliotecas más conocidas en el área médica. No solo eso, sino que posee múltiples módulos y diversos algoritmos aplicables en el estudio del área, yendo desde *machine learning* hasta métodos de preprocesamiento. La razón por la que se menciona DIPY, es porque el presente trabajo se inspiró en esta biblioteca y busca proponer una herramienta complementaria a esta, con los algoritmos desarrollados por el Grupo de Análisis

de Imágenes Médicas de nuestra Universidad. Es de importancia notar que, debido a la menor escala de este proyecto, éste carece de algunas herramientas como visualización de tractografías, por lo que se recomienda suplementar con DIPY.

2.5. PyPI - Python Packaging Index

PyPI [12] es el repositorio de software oficial para el lenguaje Python. Este permite encontrar e instalar software desarrollado y compartido por la comunidad de Python. La razón por la que se escogió PyPI es porque algunos gestores de paquetes (como *pip* o *EasyInstall*) usan PyPI como la fuente por defecto para paquetes y sus dependencias, lo que garantiza compatibilidad entre gestor de paquete y repositorio.

2.6. Doxygen

Doxygen [18] es una herramienta para generar documentación de anotaciones en códigos de C++, pero también soporta otros lenguajes, como Java, C Python. Doxygen busca anotaciones con un formato especial dentro del código, para posteriormente crear una documentación en múltiples formatos, como HTML, PDF y \LaTeX , entre otros. Se escogió Doxygen en particular por su facilidad de uso y su compatibilidad con múltiples lenguajes, considerando el caso de que tengan que documentar archivos escritos en un lenguaje distinto a Python.

2.7. Problema Encontrado

El principal problema que se identifica con los distintos códigos desarrollados por el Grupo de Análisis de Imágenes Médicas, es la dificultad su uso. Esto se debe a varios factores, los que podemos sintetizar principalmente en:

- **Falta de centralización:** Actualmente los diferentes códigos no se encuentran agrupados de ninguna manera ni en ningún repositorio. Para poder utilizarlos, se deben solicitar a los autores de cada uno o a profesores y/o estudiantes que se encuentren en posesión de estos, por lo que muchas veces se tienen que recopilar e instalar por separado.
- **Multiplidad de lenguajes:** Los códigos no están en un lenguaje común. Si bien la gran mayoría se encuentra escrito en Python, existen también códigos y bibliotecas tanto en C como en C++. Esto resulta en distintos requerimientos dependiendo del código que se quiera ejecutar. Por ejemplo,

el uso de Python requiere la instalación de una versión de Python. C y C++, por otro lado, exigen un compilador. Si bien la mayoría de las distribuciones incluyen estos lenguajes de programación, existen algunas que no. Además existen otros sistemas operativos, como Windows, que no incluyen ninguno de los dos.

No solo eso, sino que algunos de los códigos en Python llaman a funciones de los códigos y bibliotecas escritos en C o C++. Esto significa que además de tener que cumplir con los requerimientos de todos estos lenguajes simultáneamente, es necesario compilar previa y manualmente los códigos en C y C++. Esto debido a que, para poder ejecutar los programas en C/C++, es necesario compilarlos. Sin embargo, los códigos de Python que llaman a archivos en otros lenguajes no compilan los archivos que llaman de manera automática antes de ejecutarlos, por lo que la tarea termina siendo responsabilidad del usuario.

- **Pluralidad de dependencias:** Continuando con el punto anterior, así como los códigos de C y C++ pueden depender de bibliotecas, muchas veces escritas en el mismo lenguaje, los códigos de Python pueden tener dependencias de paquetes de Python, los cuales generalmente se instalan mediante gestores de paquetes como pip o Conda. El problema reside en que, los códigos no incluyen una lista de las dependencias necesarias para la ejecución de estos, ni tampoco se instalan automáticamente junto con los programas que necesitan de ellos.
- **Falta de acceso a datos de prueba:** Los códigos requieren de archivos de entrada para su ejecución, sin embargo, los códigos no necesariamente incluyen datos de prueba para poder experimentar con ellos. Además, cuando sí se incluyen, muchas veces estos no son amigables de usar, ya sea por su gran tamaño, enorme cantidad de fibras o por un extenso tiempo de ejecución.
- **Carencia de documentación:** Debido a que no existe una documentación formal para ninguno de estos códigos las únicas formas de aprender su uso son:
 1. Recibiendo las instrucciones de uso por medio de los autores de los códigos u otros usuarios con previa experiencia.
 2. Leyendo los códigos para comprender qué hacen, qué argumentos se piden, qué significa cada uno y en qué orden se ingresan.

Algunos códigos tienen comentarios en sus líneas, facilitando su entendimiento, pero no todos. Asimismo, los programas poseen varios códigos para su funcionamiento, por lo que habría que buscar cuál de todos es el principal que invoca a los otros.

Debido a que deben recopilarse e instalarse por separado. Ejecutar estos códigos también es complejo, pues muchos están escritos en distintos lenguajes de programación (Python, C, C++), lo que hace su ejecución aún más engorrosa, puesto que los códigos en C y C++ requieren la compilación de sus bibliotecas primero. Además, es necesario tener datos a mano, pues son parte de los argumentos utilizados por estos programas, para poder siquiera ejecutar los códigos. Finalmente, debido a que no existe una documentación formal para ninguno de estos códigos, las únicas formas de aprender como se usan son:

1. Recibiendo las instrucciones de uso por medio de los autores de los códigos u otros usuarios con previa experiencia.
2. Leyendo los códigos para comprender qué argumentos se piden, qué significa cada uno y en qué orden se ingresan.

Todos estos factores por separado complican mucho el uso de cada uno de estos códigos, más aún en conjunto.

3. Modelo Desarrollado

Con el objetivo de dar solución a los problemas presentados anteriormente, se desarrolló la biblioteca **Phybers**. Las características que definen a esta son:

- Desarrollada en Python, con el propósito de subir el paquete a PyPI (*Python Package Index*, [12]), facilitando su distribución y permitiendo la instalación de esta mediante gestores de paquetes de Python, como pip [13].
- Agrupa los códigos de los algoritmos en un mismo paquete, lo que en conjunto con el punto anterior, busca solucionar el problema de accesibilidad.
- Contiene las dependencias necesarias para la ejecución de cada código.
- Incluye datos de prueba en cada algoritmo para que los usuarios puedan comenzar a familiarizarse con su uso inmediatamente después de instalarla.
- Introduce una documentación para cada algoritmo, con la intención de facilitar el entendimiento de los códigos al proveer una breve descripción de cada uno, de sus argumentos y lo que significan, e instrucciones de cómo usarlos.
- Los algoritmos se dividen en módulos (los que a su vez pueden dividirse en sub-módulos), con la finalidad de que cada uno tenga sus datos de prueba y documentación asociados en el mismo módulo.
- Permite ejecutar cada algoritmo instalado al pasarle su módulo respectivo al intérprete de Python.
- Los archivos escritos en C y C++ se compilan automáticamente al llamar los módulos que requieran de ellos.

El desarrollo de la biblioteca se puede dividir en dos partes. La primera y la más extensa, sería la creación del paquete. Esto incluye la recopilación, adaptación y modularización de los algoritmos, sus códigos y sus datos de prueba asociados. La segunda parte es la documentación de los códigos. Si bien la biblioteca en sí no se encuentra dividida en códigos y documentación, sino que la documentación va incluida en su módulo respectivo, se escogió separar en dos partes para explicar en distintas secciones lo que es código más empaquetado y lo que es documentación.

La estructura jerárquica del paquete se divide en 4 módulos: *ffclust*, *hclust* (que es el módulo de clustering jerárquico), *segmentation* y *utils*. Este último a la vez, se

divide en otros 4 sub-módulos: *sampling*, *intersection*, *deform* y *postprocessing*. En la figura se puede observar una representación gráfica de la biblioteca.

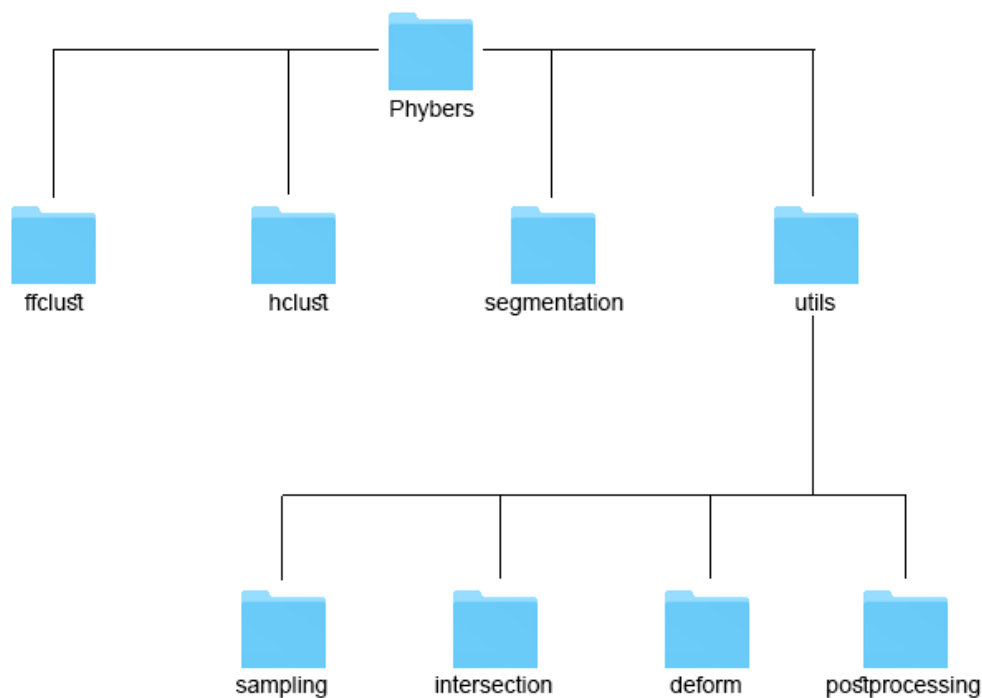


Figura 3.1: Estructura jerárquica de la biblioteca Phybers

3.1. Desarrollo del Paquete

En esta sección se explica el trabajo que se realizó con los archivos para poder empaquetarlos y posteriormente, subirlos a PyPI.

3.1.1. Preparación

Para la preparación del trabajo se instaló una partición de *Manjaro* [14], una distribución del sistema operativo Linux. Se prefirió una instalación del sistema operativo en una partición separada del disco duro por sobre una en una máquina virtual, por las siguientes razones:

- **Más recursos en el sistema operativo:** Cuando un sistema operativo *host* (quien ejecuta el software para hospedar máquinas virtuales) corre máquinas virtuales, este debe entregar recursos a la máquina virtual para que el sistema operativo *guest* (sistema operativo de la máquina virtual hospedada) pueda funcionar.

Si bien la asignación de recursos es definida por el usuario, no deja de ser un problema el dividir recursos entre dos sistemas operativos, puesto que hace más lenta la ejecución de los códigos. Además, si no se asigna la cantidad correcta de recursos (lo que depende de los procesos que se han de ejecutar en la máquina virtual), muchas veces las máquinas virtuales se congelan o los códigos dejan de ejecutarse abruptamente al no tener suficiente memoria o capacidad de procesamiento para seguir ejecutando instrucciones.

En contraste con lo mencionado, utilizar un único sistema operativo (en el caso de una partición separada), asigna todos los recursos del computador al mismo ambiente de trabajo, por lo que los problemas anteriores dejan de estar presentes.

- **Facilidad de traspaso de datos:** Debido a que la mayoría de los archivos recopilados se guardaron en una unidad de almacenamiento externa, el traspasar los datos a una máquina virtual resultó ser mucho más complicado de lo esperado, debido a que estas no parecían detectar el disco duro externo. Asimismo, la gran cantidad de datos que se trabajaban, combinado con la división de recursos entre sistemas operativos, resultan en mayores tiempos de traspaso y lectura de archivos, lo que se traduce en una menor eficiencia del trabajo y en tiempos más altos de ejecución.

Luego se instaló Python 3, pues se escogió como el lenguaje de preferencia para el desarrollo del diseño presentado. Debido a que venían incluidos en la distribución, no se tuvo que realizar ninguna instalación de los compiladores *gcc* y *g++*. Sin embargo, es importante destacar que su uso es requerimiento para la compilación y ejecución de los módulos desarrollados.

Posteriormente, se recopilaron los códigos de los algoritmos *FFClust*, Clustering Jerárquico, Segmentación y el de *UtilsTools*, funciones que calculan múltiples métricas asociadas a los fascículos. Para esto, se acudió a alumnos tesis de pregrado y postgrado del Departamento de Ingeniería Eléctrica de la Universidad de Concepción, que se encontraban en posesión de las versiones más actualizadas

de los códigos y sus respectivos datos de prueba, necesarios para poder ejecutar los códigos y probar su correcto funcionamiento.

Una vez recopilados y testeados, los archivos se dividieron en 4 carpetas, cada una correspondiente a uno de los 4 módulos mencionados previamente. Con esto, se construyó localmente la estructura de los archivos que componen el paquete.

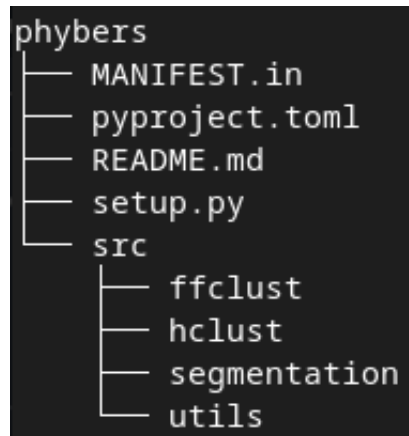


Figura 3.2: Estructura de la última versión local del paquete *phybers*.

En la figura 3.2 se puede observar que `src` es la carpeta en la que van los módulos del proyecto. El archivo `pyproject.toml` es un archivo que contiene los requerimientos del sistema para poder construir el paquete. Por otro lado, el archivo `README.md` contiene la descripción del paquete a aparecer en la página de *PyPI*. A continuación, se explica la función del resto de archivos.

3.1.2. Archivo *setup.py*

El archivo `setup` es el archivo de configuración para *setuptools* [15], una biblioteca que facilita la distribución e instalación de paquetes de python. `setup.py` posee los metadatos del paquete, por lo que le entrega a *setuptools*, información como el nombre del paquete, su versión y/o las dependencias necesarias a incluir en la instalación del paquete. Existen dos tipos de metadatos para usar en el archivo `setup` [16]:

- Estática (**setup.cfg**): Garantiza el mismo resultado siempre. Más sencillo de leer y evita algunos errores.

- Dinámica (**setup.py**): Posiblemente no determinista. Objetos no dinámicos o determinados en tiempo de instalación, tienen que ir en `setup.py`.

Para este trabajo, se escogió un archivo `setup` de tipo dinámico, pues se alinea mejor con los requerimientos e ideas del proyecto. Se puede observar como quedó la versión final del archivo `setup` en la figura 3.3.

```
1 import setuptools
2 import os
3 import subprocess
4 from pathlib import Path
5
6 this_directory = Path(__file__).parent
7
8 long_description = (this_directory / "README.md").read_text()
9
10 setuptools.setup(name='phybers',
11                  version='0.0.1',
12                  description='Integration of multiple tractography and neural-fibers related tools.',
13                  long_description=long_description,
14                  long_description_content_type='text/markdown',
15                  url='https://github.com/GonzaloSabat/MT',
16                  author='Gonzalo Sabat',
17                  author_email='gsabat@udec.cl',
18                  license='UdeC',
19                  package_dir={"": "src"},
20                  packages=setuptools.find_packages(where="src"),
21                  include_package_data=True,
22                  install_requires=[
23                      'numpy',
24                      'dipy',
25                      'joblib',
26                      'matplotlib',
27                      'sklearn',
28                      'networkx',
29                      'pandas'
30                  ]
31 )
```

Figura 3.3: Contenido del archivo `setup.py` de la biblioteca.

3.1.3. Archivo *MANIFEST.in*

Como se mencionó al principio de la sección 3, el paquete incluye datos de prueba con el fin de que este sea usable inmediatamente tras instalarse. Sin embargo, para poder incluir los archivos extra de cada paquete, es necesario escribir un archivo `MANIFEST.in` en la raíz del proyecto que especifique los archivos que se desean añadir (o remover archivos incluidos por defecto) a la distribución, puesto que solo un grupo mínimo de archivos (definidos por ciertos patrones) son incorporados por defecto.

Este archivo mediante ciertos comandos, le da la instrucción a *setuptools* para

que incluya o excluya grupos de archivos de la distribución. Por ejemplo, al escribir lo siguiente en el archivo `MANIFEST.in`

```
include src/ffclust/data/*.bundles
include src/ffclust/data/*.bundlesdata
```

setuptools adjunta a la distribución todos los archivos que se encuentren en el directorio `src/ffclust/data/` y que tengan como extensión `.bundles` o `.bundlesdata`. Esto también se puede expandir a más casos y extensiones. En el caso del paquete, se usó para incluir archivos con extensión `.jpg` de ciertas imágenes usadas en los algoritmos, archivos `.h`, extensión de ciertas bibliotecas que ocupan algunos códigos escritos en C o incluir directorios enteros de manera recursiva, entre otros usos.

Dado que el paquete está compuesto por varios módulos y cada uno de estos necesita incluir distintos archivos para su correcta ejecución, se detalló manualmente lo necesario para cada módulo (Ver Figura 3.4).

```
1 include src/segmentation/*.cpp
2 include src/segmentation/*.py
3 recursive-include src/segmentation/data/*
4 graft src/segmentation/data
5
6
7 include src/ffclust/*.py
8 include src/ffclust/*.jpg
9 include src/ffclust/*.txt
10 include src/ffclust/data/*.bundles
11 include src/ffclust/data/*.bundlesdata
12 include src/ffclust/data/ffclust/*.cpp
13 include src/ffclust/data/ffclust/*.txt
14 include src/ffclust/FibersTools/*.py
15 recursive-include src/ffclust/segmentation_clust_v1.1/*
16 graft src/ffclust/segmentation_clust_v1.1
17
18
19 include src/hclust/*.py
20 include src/hclust/hierarchical/*.cpp
21 include src/hclust/hierarchical/*.c
```

Figura 3.4: Extracto del contenido del archivo MANIFEST.

3.1.4. Modularización de los códigos

Debido a que uno de los objetivos propuestos es subir el paquete a PyPI, es necesario modificar los archivos de manera que un usuario pueda usarlos después de su instalación. Es importante, por lo tanto, modularizar los códigos, lo que implica separar las funcionalidades de estos en módulos independientes o en distintos sub-módulos. Para efectos del trabajo, la modularización se alcanzó mediante 4 ejes principales:

1. Identificando el código principal (que realiza el llamado a otros códigos) y modificándolo para ejecutar su contenido al correr el módulo como script. Esto se logró mediante el archivo `__main__.py`.
2. Portabilizando los códigos, cambiando los llamados a rutas relativas por rutas absolutas de manera dinámica al utilizar la variable `__file__`.
3. Garantizando el funcionamiento de las bibliotecas y códigos requeridos al ejecutar un módulo. Esto se consiguió con la función `run` del módulo `subprocess`.
4. Permitiendo al usuario ingresar los argumentos por consola, en vez de modificar el código fuente. Para esto se utilizó la lista `sys.argv[]`.

1) `__main__.py` Para lograr el punto 1, se aprovechó el archivo intrínseco de paquetes de Python, `__main__.py`. Este archivo provee una interfaz de línea de comandos para un paquete, de manera que podamos llamar directamente a un módulo. Por ejemplo, supongamos que tenemos un paquete de nombre "*example*", cuya estructura jerárquica y contenido del archivo `__main__.py` se pueden ver en las figuras 3.5 y 3.6, respectivamente:

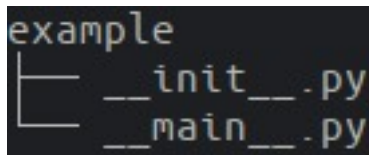


Figura 3.5: Estructura del paquete local *example*.

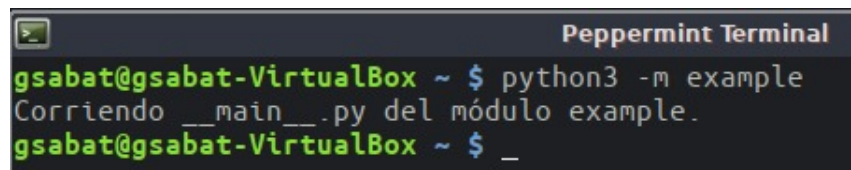
```
1 print("Corriendo __main__.py del módulo example.")
2
```

Figura 3.6: Contenido del archivo `__main__.py` del paquete *example*.

Para ejecutar el módulo *example* desde la línea de comandos, basta con agregar el argumento `-m`, lo que permite que el módulo se ejecute como si fuera un programa. Luego, ingresar la siguiente línea en la shell:

```
$ python3 -m example
```

Ejecutará el archivo `__main__.py` del módulo "example", entregando como salida lo siguiente (Ver Figura 3.7):



```
Peppermint Terminal
gsabat@gsabat-VirtualBox ~ $ python3 -m example
Corriendo __main__.py del módulo example.
gsabat@gsabat-VirtualBox ~ $ _
```

Figura 3.7: Ejecución del archivo `__main__.py`.

De esta manera, se añadió un archivo `__main__.py` para tres de los módulos principales (*ffclust*, *hclust* y *segmentation*) y para cada uno de los 4 sub-módulos del módulo *utils*, con el propósito que cada módulo pueda ser llamado individualmente. El contenido de estos archivos es una versión del archivo principal de cada algoritmo, modificada para implementar los otros dos ejes necesarios para poder modularizar el código.

2) Atributo `__file__` En programación existen dos tipos de rutas:

1. Ruta absoluta: Es la ruta completa, desde el directorio raíz (por lo que siempre comienzan en `/`) hasta el archivo o carpeta que se desea, independiente del directorio actual en que uno se encuentra. Un ejemplo de ruta absoluta sería, por ejemplo:

```
/home/gsabat/.local/lib/python3.6/site-packages/example
```

Sin importar en donde nos encontremos en la línea de comandos actualmente, la ruta absoluta al módulo *example* siempre será la misma.

2. Ruta relativa: Es una ruta que empieza desde el directorio en que se está trabajando, por lo que no es necesario proveer la ruta completa. Siguiendo con el mismo ejemplo presentado anteriormente, si nuestro directorio actual de trabajo fuera: `/home/gsabab/.local/lib/python3.6` entonces, la ruta relativa al mismo módulo *example* sería:

`/site-packages/example`

Al tener los dos tipos de rutas, es importante preguntarse: si los códigos se movieran a un computador distinto (instalándolos mediante *PyPI*) y se necesitara acceder a los archivos, ¿Dónde estarían esos archivos? ¿Desde dónde ejecutaría el usuario el script?. Puesto que es mejor asumir el peor caso (no se sabe desde donde ejecutará el script el usuario y el usuario no sabe dónde se instalarán los archivos), se propuso computar la ruta absoluta en el sistema del usuario, mediante el uso en conjunto de funciones del módulo `os` y la variable `__file__`. `__file__` es una variable global de cada script de Python, que devuelve la ruta relativa al archivo `.py` que la contiene. Al entregarle esta variable como argumento a la función `path.dirname()` del módulo `os`, se pudo obtener una ruta absoluta al directorio que contiene al script ejecutado en cuestión.

Por ejemplo, supongamos que un usuario desea ejecutar el módulo *segmentation*, cuya ruta post-instalación es `/home/gsabab/.local/lib/python3.6/site-packages/segmentation` y que este posee la siguiente estructura (Ver Figura 3.8):

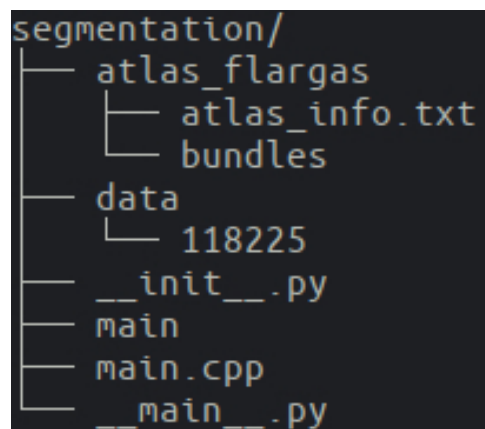


Figura 3.8: Estructura jerárquica del módulo *segmentation*.

Además, el contenido del archivo `__main__.py` de *segmentation* es el siguiente:

```
1 from subprocess import run
2
3 npoints = "21"
4 fibrasdir = "data/118225/118225.bundles"
5 idsubj = "118225"
6 atlasdir = "atlas_flargas/bundles"
7 atlasInformation = "atlas_flargas/atlas_info.txt"
8 seg_result = "seg_result"
9 id_seg_result = "idseg_result"
10
11 run(["./main", npoints, fibrasdir, idsubj, atlasdir, atlasInformation,
      seg_result, id_seg_result])
```

Se puede observar que ciertas variables, como las definidas en las líneas 4, 6, 7, 8 y 9, equivalen a una ruta relativa a ciertos archivos y directorios (puesto que no comienzan con `/`, el directorio raíz) que se usan en la línea 11 como argumentos para el archivo ejecutable `main`, el cual llama la función `run()` (la cual se explicará en la siguiente sección). El problema entonces, es que cuando un usuario intente ejecutar el módulo *segmentation*, las rutas relativas de estas variables se verán precedidas por el directorio en que se encuentre el usuario. Si el usuario ejecuta `python3 -m segmentation`, mientras que su directorio actual de trabajo es el directorio raíz (`/`), entonces el código, no solo asignará una ruta relativa incorrecta a las variables (debido a que el directorio de trabajo actual no es el del módulo *segmentation*), sino que ni siquiera podrá ejecutar `./main`, pues el archivo ejecutable no se encuentra en el directorio raíz.

Sin embargo, este problema se solucionó al asignarle a una variable el valor retornado por `os.path.dirname(__file__)`. Esta función retorna la ruta absoluta y el nombre del directorio en que se encuentra el código que utiliza esta función. Modificando el archivo `__main__.py`, que se presentó previamente de la siguiente manera:


```

1 from subprocess import run
2 import os
3
4 pathname = os.path.dirname(__file__)
5
6 npoints = "21"
7 fibrasdir = pathname + "data/118225/118225.bundles"
8 idsubj = "118225"
9 atlasdir = pathname + "atlas_flargas/bundles"
10 atlasInformation = pathname + "atlas_flargas/atlas_info.txt"
11 seg_result = pathname + "seg_result"
12 id_seg_result = pathname + "idseg_result"
13
14 run([pathname + "/./main", npoints, fibrasdir, idsubj, atlasdir,
      atlasInformation, seg_result, id_seg_result])

```

Como *pathname* guarda la ruta absoluta del directorio perteneciente al módulo *segmentation*, al concatenar *pathname* con el resto de la ruta a los archivos que se desean utilizar como argumentos, se obtiene la ruta absoluta a cada uno de estos archivos. La idea es que la mayoría de los argumentos de los módulos se ingresen por consola, por lo que no serviría concatenar *pathname* a algunas de estas variables. Sin embargo, esto resuelve el problema de no poder llamar al archivo ejecutable *main* desde cualquier directorio. La solución propuesta también funciona para archivos como las bibliotecas (archivos *.h*) que se encuentren dentro del módulo que sea necesario compilar para su funcionamiento y por tanto, también sirve para el archivo de salida de la compilación de estos.

De este modo, se logró la portabilidad de los códigos y se garantizó poder llamar a los módulos independientemente del directorio actual de trabajo del usuario.

3) subprocess.run(): Para alcanzar el tercer eje de la modularización, se necesitaba garantizar la existencia y ejecución de ciertos archivos; particularmente los archivos de salida generados al compilar archivos con extensiones `.h`, `.c` y `.cpp`. Ante esto, se propuso compilar automáticamente dichos archivos al ejecutar el módulo correspondiente y revisar que los ficheros de salida se hayan creado correctamente. Para lograr esto, se utilizó el módulo *subprocess*. Este módulo permite crear nuevos procesos, conectar a sus tuberías de entrada/salida/error y obtener sus códigos de retorno. Específicamente, la función que se implementó del módulo fue `run()`. Se escogió esta función en vez de alternativas como `call()`, pues es el enfoque recomendado para todos los casos de uso que puede soportar, como se menciona en la documentación del módulo [17].

La función `run()` permite correr un comando y sus parámetros, descritos por los argumentos entregados a esta. Esto prueba ser muy útil para ejecutar funciones que normalmente se harían mediante el usuario por medio de consola. En combinación con los ejes anteriores, se puede utilizar `run()` para compilar automáticamente ciertos archivos al ejecutar el archivo `__main__.py` de un módulo o correr otros programas, independiente que hayan sido escritos en Python o C/C++.

Tomemos por ejemplo, el modificado archivo `__main__.py` del módulo *segmentation*, presentado en el eje 2:

```
1 from subprocess import run
2 import os
3
4 pathname = os.path.dirname(__file__)
5
6 npoints = "21"
7 fibrasdir = pathname + "data/118225/118225.bundles"
8 idsubj = "118225"
9 atlasdir = pathname + "atlas_flargas/bundles"
10 atlasInformation = pathname + "atlas_flargas/atlas_info.txt"
11 seg_result = pathname + "seg_result"
```

```

12 id_seg_result = pathname + "idseg_result"
13
14 run([pathname + "/./main", npoints, fibrasdir, idsubj, atlasdir,
      atlasInformation, seg_result, id_seg_result])

```

Se puede observar en las líneas 1 y 14 que se hace uso de la función `run()` para poder ejecutar el archivo generado al compilar el código `main.cpp`. Sin embargo, surge un problema, debido a que no se incluyen archivos ejecutables en el paquete (debido a restricciones de PyPI). Si un usuario fuera a descargarlo via *PyPI*, no podría ejecutar el archivo `main` de la línea 14, pues este archivo no existiría. Para asegurar la ejecución correcta del módulo, se utilizó `run()` para compilar el archivo `main.cpp`, en conjunto con la función `os.path.exists()`, que revisa si la ruta que se le entrega como argumento existe. Al entregarle la ruta absoluta al archivo ejecutable `main`, se puede garantizar la existencia o compilación del archivo previo a la ejecución de este, lo que aplicado al archivo `__main__.py`, resulta así:

```

1 from subprocess import run
2 import os
3
4 pathname = os.path.dirname(__file__)
5
6 npoints = "21"
7 fibrasdir = pathname + "data/118225/118225.bundles"
8 idsubj = "118225"
9 atlasdir = pathname + "atlas_flargas/bundles"
10 atlasInformation = pathname + "atlas_flargas/atlas_info.txt"
11 seg_result = pathname + "seg_result"
12 id_seg_result = pathname + "idseg_result"

```

```

13 if os.path.exists(pathname + "/main"):
14     print("Executable file found. Running executable file: ")
15 else:
16     print("Executable file not found. Compiling main.cpp...")
17     run(['g++', '-std=c++14', '-O3', pathname + '/main.cpp', '-o',
        pathname + '/main', '-fopenmp', '-ffast-math'])
18     if os.path.exists(pathname + "/main"):
19         print("main.cpp compiled. Running executable file: ")
20     else:
21         print("Executable file still not found. Exiting...")
22         exit()
23
24
25 run([pathname + "/./main", npoints, fibrasdir, idsubj, atlasdir,
    atlasInformation, seg_result, id_seg_result])

```

El código añadido (encontrado desde la línea 13 a la 22) al archivo, revisa si existe el archivo ejecutable main, es decir, busca en el módulo *segmentation* si se ha compilado alguna vez main.cpp. Si lo encuentra, simplemente lo ejecuta con sus parámetros correspondientes. En el caso contrario, la función run() compila automáticamente el archivo main.cpp, como se puede observar en la línea 17. Finalmente, el programa realiza otro chequeo, si el archivo sí se encuentra ahora, lo ejecuta. De no encontrarse nuevamente el ejecutable, el programa arroja un mensaje de error y termina el programa.

Es de importancia destacar que para todos los módulos incluidos que ejecutan códigos en C/C++ o usan bibliotecas en que se implementó el procedimiento presentado, solo se compilan la primera vez que se ejecuta el módulo o cada vez que no se encuentre el archivo de salida en cuestión (lo que no debería pasar sin intervención manual de los archivos).

4) `sys.argv[]`: El último procedimiento que se realizó para modularizar los códigos fue modificarlos de forma que el usuario escogiera, al entregar parámetros por consola, los argumentos y archivos que se utilizarán en la ejecución de un módulo. Si bien cuando los códigos se recopilaron, estos venían con rutas relativas a datos de prueba, si un usuario quisiera probar con datos propios, tendría que modificar el código fuente para que las variables guarden las rutas equivalentes a los nuevos archivos. Por la falta de practicidad de esto, se decidió utilizar la lista `sys.argv[]`.

El módulo `sys` provee funciones y variables que se usan para manipular distintas partes del ambiente en tiempo de ejecución de Python. `sys` proporciona acceso a algunas variables usadas por el intérprete. La lista `argv[]`, guarda los argumentos de la línea de comandos en el mismo orden que se entregan al programa, donde `argv[0]` siempre es el nombre del script que se está ejecutando. Ajustando el archivo `__main__.py` del módulo *segmentation* presentado anteriormente, para implementar `sys.argv[]`, se obtiene:

```
1  from subprocess import run
2  import os
3  import sys
4
5  pathname = os.path.dirname(__file__)
6
7  npoints = sys.argv[1]
8  fibrasdir = sys.argv[2]
9  idsubj = sys.argv[3]
10 atlasdir = sys.argv[4]
11 atlasInformation = sys.argv[5]
12 seg_result = sys.argv[6]
13 id_seg_result = sys.argv[7]
```

```

14 if os.path.exists(pathname + "/main"):
15     print("Executable file found. Running executable file: ")
16 else:
17     print("Executable file not found. Compiling main.cpp...")
18     run(['g++', '-std=c++14', '-O3', pathname + '/main.cpp', '-o',
19         pathname + '/main', '-fopenmp', '-ffast-math'])
19     if os.path.exists(pathname + "/main"):
20         print("main.cpp compiled. Running executable file: ")
21     else:
22         print("Executable file still not found. Exiting...")
23         exit()
24
25
26 run([pathname + "/./main", npoints, fibrasdir, idsubj, atlasdir,
27     atlasInformation, seg_result, id_seg_result])

```

De esta manera, cuando el usuario desee ejecutar algún módulo del paquete, deberá ingresar como argumentos del programa, las variables y rutas absolutas de archivos que desee usar.

Así entonces, se puede observar el proceso que se empleó de manera general para cada uno de los módulos implementados. La conjunción de los 4 ejes implementados, fue suficiente para modularizar la mayoría de los códigos. Sin embargo, esto no significa que haya sido el único trabajo que se realizó en los módulos. A continuación, se presentan los módulos implementados junto con su entrada y salida.

3.1.5. Módulo *segmentation*



Figura 3.9: Demostración gráfica de entrada y salida del módulo *segmentation*.

En la figura 3.9, se presenta la entrada (números decimales) y salida (números romanos) del módulo *segmentation*, en donde:

1. **npoints**: Es el número de puntos a los que se hace la segmentación.
 2. **fibrasdir**: Ruta al archivo de los fascículos.
 3. **idsubj**: ID del sujeto a quien pertenecen los fascículos.
 4. **atlasdir**: Ruta al directorio que contiene los archivos `.bundles` y `.bundlesdata` del atlas.
 5. **atlasInformation**: Ruta al archivo `.txt` que contiene la información del atlas.
 6. **result_path**: Ruta al directorio de salida para los fascículos segmentados para el sujeto, en donde se crearán dos directorios más, uno para guardar los fascículos segmentados para el sujeto (`seg_result`) y otro para guardar los índices de las fibras originales para cada fascículo detectado (`idseg_result`).
- I Fascículos segmentados.
- II Centroides de los fascículos segmentados.
- III ID de las fibras de los fascículos segmentados.

3.1.6. Módulo *ffclust*



Figura 3.10: Representación gráfica de entrada y salida del módulo *ffclust*. Aquellos notados con * son argumentos opcionales.

En la figura 3.10, se presenta la entrada del módulo *ffclust*, en donde:

1. **points**: Cantidad de puntos definidos para la ejecución del algoritmo. Para el correcto funcionamiento del algoritmo, los puntos ingresados deben ser 0, 3, 10, 17 y 20.
 2. **ks**: Cantidad óptima de clusters k-means para cada uno de los puntos elegidos. Los índices sugeridos para los puntos mencionados anteriormente son 300, 200, 200, 200 y 300, respectivamente.
 3. **infile**: Ruta al archivo .bundles de entrada.
 4. **output-directory**: Ruta del directorio de salida del módulo.
 5. **thr-segmentation***: Es un umbral para reasignar clústers. Es de importancia notar que este argumento es opcional y en caso de no ser ingresado, su valor por defecto es 6.
 6. **thr-join***: Es un umbral para reagrupar grupos con más similitud. Es importante notar que este argumento es opcional y en caso de no ser ingresado, su valor por defecto es 6.
- I Directorio que contiene clústers detectados en bundles (**FinalBundles**).
- II Directorio que contiene centroides de los clústers detectados (**FinalCentroids**).
- III Directorio que contiene parámetros del algoritmo (**output**).
- IV Estadísticas del desempeño en tiempos de ejecución (**info.log**).

V Estadísticas sobre tiempo kmeans, map y segmentación (**stats.txt**).

3.1.7. Módulo *hclust*



Figura 3.11: Representación gráfica de entrada y salida del módulo *hclust*.

En la figura 3.11, se puede apreciar la entrada y salida del módulo *hclust*, donde:

1. **dir_raw_tractography**: Ruta del archivo .bundles de la tractografía.
 2. **MaxDistance_Threshold**: Máximo umbral de distancia euclidiana entre pares de fibras. Se recomienda 40.
 3. **maxdist**: Máximo umbral de afinidad. Se recomienda 30.
 4. **var**: Valor relacionado con la escala de similitud utilizada. Se recomienda 3600.
 5. **result_path**: Ruta del directorio de salida.
- I Archivos .bundles y .bundlesdata los clústers detectados.

3.1.8. Sub-módulo *utils.intersection*



Figura 3.12: Representación gráfica de entrada y salida del módulo *intersection*.

1. **dir_fib1**: Ruta a un fascículo .bundles.
 2. **dir_fib2**: Ruta a otro fascículo distinto, también de extensión .bundles.
 3. **outdir**: Ruta del directorio de salida de archivo .txt.
 4. **d_th**: Umbral de intersección en milímetros. Se usa con un valor alrededor de 10, pero depende de la aplicación del usuario.
- I Archivo .txt que contiene los porcentajes de intersección que explica cuantas fibras tienen en común los dos fascículos.

3.1.9. Sub-módulo *utils.deform*



Figura 3.13: Representación gráfica de entrada y salida del módulo *deform*.

1. **in_imgdef**: Imagen de formato *.nii* que contiene la deformación para transformar las fibras para el espacio MNI en sujetos de la base de datos HCP.
 2. **indir**: Ruta de archivo de la fibra a transformar.
 3. **outdir**: Ruta del directorio de salida.
- I Fibra transformada en espacio MNI.

3.1.10. Sub-módulo *utils.sampling*



Figura 3.14: Representación gráfica de entrada y salida del módulo *sampling*.

1. **indir**: Ruta de fascículos *.bundles* de entrada.
2. **in_npoints**: Cantidad de puntos.
3. **outdir**: Ruta de directorio de salida.

| Archivo **.bundles** muestreado a **n** puntos.

3.1.11. Sub-módulo *utils.postprocessing*



Figura 3.15: Representación gráfica de entrada y salida del módulo *postprocessing*.

1. **in_statisticsPath**: Ruta de directorio de entrada que contiene resultados de un algoritmo de segmentación o alguno de clustering. Actualmente funciona solo sobre FFClust.
- | Directorio con distintas métricas, incluyendo un objeto *pandas*.

3.2. Documentación

Para el proceso de documentación, se escogió *Doxygen* [18], una herramienta para generar documentación de manera automática mediante anotaciones (en forma de comentarios) en el código. Doxygen soporta múltiples lenguajes de programación populares, incluyendo C, C++ y Python, que son los lenguajes existentes en el paquete.

Luego, se anotaron comentarios dentro del código `__main__.py` de cada módulo. Como Doxygen detectó automáticamente las variables de estos, se escribió un comentario una línea arriba de cada variable, específicamente de aquellas que representan argumentos ingresados por el usuario, describiendo brevemente el significado de cada una. En la siguiente figura (Ver Figura 3.16), se puede observar un ejemplo de la notación que se usó en el código:

```
1 ## @package segmentation.main
2 # @subsection description Como ejecutar
3 # El código main de segmentation se ejecuta de la siguiente manera:
4 #
5 # > python3 -m segmentation <npoints> <fibrasdir> <idsbj> <atlasdir> <atlasInformation> <result_path>
6 #
7 #
8 import sys
9 import os
10 from pathlib import Path
11 from subprocess import run
12
13
14 ## Variable que guarda la ruta absoluta del archivo, para poder correr el ejecutable /main. No es ingres
15 pathname = os.path.dirname(__file__)
16 ## Es el número de puntos a los que se hace la segmentación.
17 npoints = sys.argv[1]
18 ## Ruta al archivo .bundles correspondiente a los fascículos.
19 fibrasdir = sys.argv[2]
20 ## ID del sujeto a quien pertenecen los fascículos.
21 idsbj = sys.argv[3]
22 ## Ruta al directorio que contiene los archivos .bundles y .bundlesdata del atlas.
23 atlasdir=sys.argv[4]
24 ## Ruta al archivo .txt que contiene la información del atlas.
25 atlasInformation = sys.argv[5]
26 ## Directorio de salida para los fascículos segmentados del sujeto.
27 result_path = sys.argv[6]
```

Figura 3.16: Comentarios con notación de Doxygen en código de módulo *segmentation*.

Luego, se ejecutó en consola el comando correspondiente para construir la documentación en base a estas anotaciones. Al terminar de ejecutarse, Doxygen entregó múltiples archivos y carpetas, entre ellos, `index.html`, que es la página inicial de la documentación de los códigos. En la figura 3.17, se puede observar el resultado de la documentación del módulo `segmentation` en formato HTML (Ver Figura 3.17).

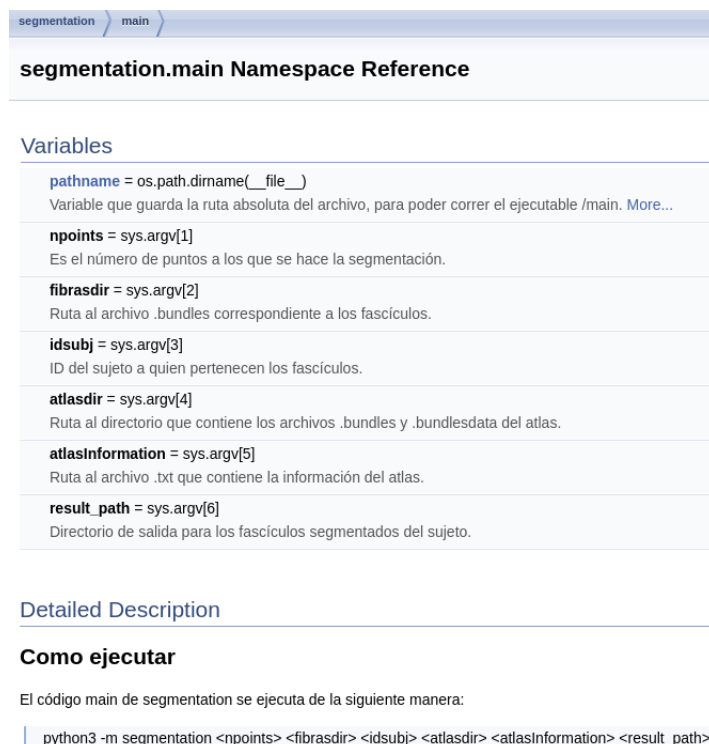


Figura 3.17: Documentación generada por Doxygen, en formato HTML.

Cabe destacar que Doxygen también genera una carpeta con archivos de \LaTeX y un archivo `Makefile`, por lo que también es posible construir una documentación en formato `.pdf`.

3.3. Subida a PyPI

Una vez finalizado el proceso de documentación e incluidos sus archivos en el paquete, se comenzó el proceso para subirlo a *TestPyPI*, una instancia separada de *PyPI* que permite probar herramientas de distribución y procesos sin afectar el verdadero índice.

Primero se tuvo que crear la distribución a subir, para lo que se abrió la consola de sistema en el directorio raíz del paquete y se ejecutó el comando `python3 -m build --sdist`. Esto crea un directorio `dist`, que contiene una distribución del paquete de extensión `.tar.gz`.

Luego, para subir el paquete de manera segura a *TestPyPI*, se creó un token API asociado a la cuenta de *TestPyPI*. La herramienta que se escogió para subir el paquete fue *twine*, debido a que es la herramienta recomendada en el tutorial que se siguió. Una vez instalado, se ejecutó *twine* para subir los archivos dentro del directorio `dist`, de la siguiente manera:

```
python 3 -m twine upload -repository testpypi dist/*
```

Al ejecutarlo, se ingresaron las credenciales requeridas (`__token__` como usuario y el valor del token API generado, como contraseña) en consola. Una vez completado el comando, el paquete ya se encontraba subido a *TestPyPI*, como se puede observar en la siguiente figura:

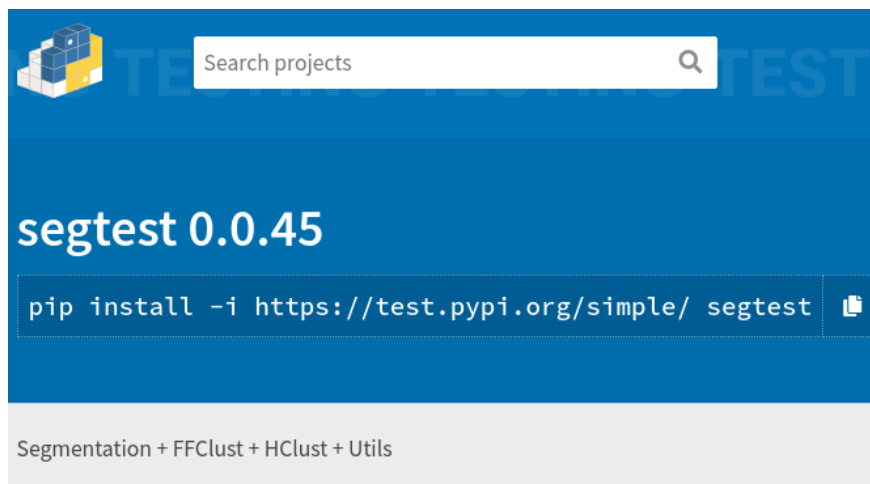


Figura 3.18: Versión 0.0.45 del paquete *segtest* en el sitio de PyPI.

Como se puede observar, el paquete mostrado en la figura 3.18 no es *phybers*, sino *segtest*. Sin embargo, *segtest* es el nombre que tuvo el paquete durante su desarrollo, previo a la decisión del nombre *phybers*. Cuando el paquete alcanzó su versión final de desarrollo, se cambió el nombre a *phybers* y se subió al verdadero PyPI.

4. Experimentos y Resultados

En esta sección, se describen los experimentos realizados durante el desarrollo y sus resultados.

Los experimentos realizados consistieron en probar el funcionamiento del paquete y sus distintas funciones desde el punto de vista de un usuario sin mayores conocimientos. Para esto, se creó una máquina virtual con el sistema operativo *Peppermint OS*, una distribución de Linux. Al crearla, se escogió como opción la más minimalista, intentando emular el peor caso posible para un usuario nuevo, es decir, que la distribución del usuario estuviera recién instalada e incluyera la menor cantidad de funcionalidades posibles. Los requerimientos para la instalación y ejecución del paquete son las siguientes:

- **gcc**
- **g++**
- **pip3**
- **python 3**

Además, de esto, los códigos incluidos poseen múltiples dependencias a otros paquetes de Python, sin embargo como se puede ver en el archivo *setup.py* (Ver sección 3.1.2), como estas dependencias están incluidas en el parámetro *install_requires* del paquete, estas se instalan en el tiempo de instalación del paquete.

Una vez creada la máquina virtual e instalado el sistema operativo de esta, se instaló **pip3** y **python3**, para poder descargar el paquete. Para descargarlo, se abrió la terminal de la máquina virtual y se ejecutó:

```
pip3 install --extra-index-url https://test.pypi.org/simple/ segtest
```

Se tuvo que incluir el argumento `--extra-index-url` porque las dependencias especificadas en *setup.py* normalmente se descargan del PyPI real. Así, al descargar el paquete desde TestPyPI, el comando intenta descargar las dependencias desde TestPyPI y por lo tanto, muchas no se encuentran o se encuentran versiones incorrectas. Cuando se descarga el paquete desde el verdadero PyPI, no es necesario incluir este parámetro o el enlace a TestPyPI como argumentos.

Una vez instalado el paquete en la ruta por defecto para pip, se probó el funcionamiento de los módulos. Para ello, cada vez que se subió una versión nueva del

paquete, se ejecutaron todos los módulos contenidos con sus datos de prueba. Actualmente, con la versión del paquete, todos los paquetes funcionan correctamente, lo que no significa que no haya espacio para mejoras. A continuación, se presentarán algunos de los problemas encontrados en los distintos módulos durante la experimentación, cómo se solucionaron (o que habría que hacer para solucionarlo en un futuro) y ciertas ideas que surgieron en el proceso.

El primer problema encontrado fue al intentar crear una distribución *wheel* del paquete, que incluyera los datos de prueba. Existen dos tipos de distribución de paquete:

- **source** (.tar.gz): Formato que provee metadatos y los archivos fuente esenciales necesarias para instalar con una herramienta como pip. El resultado es un archivo que puede ser usado para recompilar todo en cualquier plataforma como Linux, Windows y Mac.
- **wheel** (.whl): Formato de paquete que permite instalar software más rápidamente. Cuando se crea una distribución .whl, esta se crea para una combinación específica de sistema operativo y versión de Python.

Al intentar instalar un paquete con pip, no es necesario especificar que se quiere instalar una distribución *wheel*, pues pip siempre intentará instalar una distribución *wheel* por defecto, a menos que no encuentre un archivo .whl para el sistema operativo deseado, caso en que se instalará con la distribución *source*.

El problema surgió al intentar incluir datos de prueba en el paquete para el módulo *segmentation*. Debido a que el archivo `MANIFEST.in` es específicamente para distribuciones *source*, la distribución *wheel* no incluía los datos de prueba. Se intentó añadir los datos de prueba para el .whl en el archivo `setup.py`, pero actualmente, existen múltiples parámetros para incluir datos extra a un .whl. Se probaron algunos de estos parámetros para poder construir una distribución *wheel*, sin embargo, luego de varios intentos fallidos, se optó por construir solo una distribución *source*, pues los archivos extra ya se habían adjuntado correctamente a esta distribución.

4.1. *Segmentation*

Durante la experimentación del módulo *segmentation*, no se enfrentaron mayores dificultades con los tiempos de ejecución o con los datos. Sin embargo, con este módulo en particular, se implementaron algunas características extra:

- La opción de añadir `-s` como argumento, de manera que antes que se ejecute el algoritmo de segmentación, se ejecute el módulo de *sampling* a 21 puntos, con tal que el archivo de salida de *sampling* sea la entrada de segmentación.
- Un submódulo *example*, que se ejecuta con `python3 -m segmentation.example`, el cual ejecuta segmentación con los datos de prueba incluidos en el paquete y los guarda en un directorio definido por defecto, que luego se imprime su ruta en consola para que el usuario pueda ver los resultados.
- Un submódulo *help* que se ejecuta con `python3 -m segmentation.help`. Al ejecutarse, el programa imprime en consola los argumentos y el orden en que se deben ingresar al ejecutar *segmentation* junto con un ejemplo de texto que, al copiarlo y pegarlo en consola, permite al usuario ejecutar el módulo *segmentation* con los datos de prueba incluidos. Se diferencia del módulo *segmentation.example* en que, de querer, el usuario puede modificar a su gusto los argumentos ingresados antes de ejecutar el comando.

4.2. *FFclust*

Algunos problemas que se pueden destacar durante la experimentación de *ffclust* son:

- Los datos de prueba recibidos en un principio hacían que el módulo tardara alrededor de 9 - 12 minutos en terminar de ejecutarse. Para solucionarlo, se tuvieron que solicitar datos con menos fibras, con el fin de acelerar un poco la ejecución. Al reemplazar los datos iniciales de prueba con los nuevos en el paquete, una ejecución de prueba del módulo pasó de tardarse alrededor de 10 minutos, a algunos segundos.
- Los datos de prueba recopilados inicialmente ocupaban mucho espacio de almacenamiento, por lo que, debido a limitaciones de PyPI, no se podían incluir en el paquete. Al solucionar el problema anterior, se logró conseguir un set de datos más ligero y por lo tanto, se pudo añadir al paquete.

Durante el proceso de experimentación, se intentó ejecutar el módulo de *post-processing* de manera automática tras terminar la ejecución de *ffclust*. Esto se logró pero, finalmente se optó por dejar la ejecución de *ffclust* aislada del módulo *postprocessing*.

4.3. *Hclust*

El principal problema encontrado durante la experimentación de *hclust* fue que los datos de prueba recopilados inicialmente ocupaban mucho espacio, por lo que no se podían incluir al subir el paquete a PyPI. Además, la ejecución del módulo con estos datos de prueba demoraba más de 1 hora, lo que hizo el proceso de desarrollo y experimentación del módulo mucho menos interactivo, al tener que esperar tanto tiempo para poder solucionar errores. Si bien el problema del espacio se logró resolver con nuevos datos, el tiempo de ejecución sigue siendo bastante más alto de lo deseable (alrededor de 40 minutos).

4.4. *Utils*

Durante la experimentación del módulo *utils*, los problemas e ideas que se pueden destacar son:

- Un problema que se encontró con el módulo *sampling* es que se cerró múltiples veces durante su ejecución. Durante su experimentación, se concluyó que esto se debe a problemas de asignación de memoria, dado el error entregado por consola. Una vez se probó con una entrada con menor cantidad de fibras, el algoritmo se ejecutó completamente. Sin embargo, el problema de asignación de memoria persiste para archivos con una gran cantidad de fibras o computadoras con menos memoria (como es el caso de la máquina virtual con que se experimentó).
- Un problema encontrado con el módulo *intersection* fue que, al cambiar la entrada desde rutas definidas en el código a una entregada por el usuario vía consola, la variable de umbral de intersección (*d_th*), se guardaba como *string*. Sin embargo, para poder realizar una comparación de variables encontrada en el código, *d_th* debía ser de tipo *int*. Al aplicarle una conversión de *string* a *int*, el problema se solucionó.
- Para prevenir errores al ejecutar el módulo *deform*, debido a que el primer argumento debe ser un archivo de extensión *.nii*, se agregó la restricción acorde, imprimiendo un mensaje en pantalla con las instrucciones y deteniendo la ejecución. Aparte de lo mencionado, el módulo *deform* funcionó correctamente sin problemas durante su experimentación.
- El módulo de *postprocessing* arrojó errores muchas veces al ejecutarse. Esto se debía a que, uno de los programas llamados por este módulo, creaba

varios archivos, los cuales eran creados en un directorio. Este directorio es recibido como argumento por otro programa, que leía los archivos dentro para crear un archivo *pandas*; pero no encontraba un archivo en particular. Esto resultó ser porque, el programa que creaba el directorio con los archivos dentro, creaba este archivo con un nombre, pero el otro programa lo buscaba por otro nombre. Una vez se arregló la inconsistencia, el programa funcionó de manera correcta.

- Otra observación del módulo *postprocessing*, es que, como se mencionó en la sección 3.1.11, este solo funciona aplicado sobre la salida de *ffclust*. Esto ocurre pues *postprocessing* recibe un directorio como entrada, sobre el que se aplica el post-procesamiento, accediendo a los archivos dentro del directorio de entrada por medio de rutas relativas a este. El problema es que actualmente, *postprocessing* solo funciona con la estructura de salida del módulo *ffclust*, razón por la cual no se puede aplicar a la salida de los otros módulos.

4.5. Documentación

Un problema que se encontró con la documentación fue que debido a que *Doxygen* tiene problemas para detectar la documentación del contenido de archivos como `__main__.py` o `__init__.py`, poder generar la documentación en un principio fue muy complejo, puesto a que el archivo principal de cada módulo no era compatible con *Doxygen*. Por esto, se optó por crear un archivo secundario de nombre `main.py`, con la finalidad de que *Doxygen* lo detectara, que tuviera las mismas variables que el archivo original y por lo tanto, para efectos de crear la documentación, se pusieron las anotaciones y definiciones de las variables en estos archivos auxiliares. Aun así, los archivos `__main__.py` principales, se utilizaron para poder construir la página de inicio de la documentación, que redirige a la documentación sobre el archivo `__main__.py` y sus argumentos.

Phybers

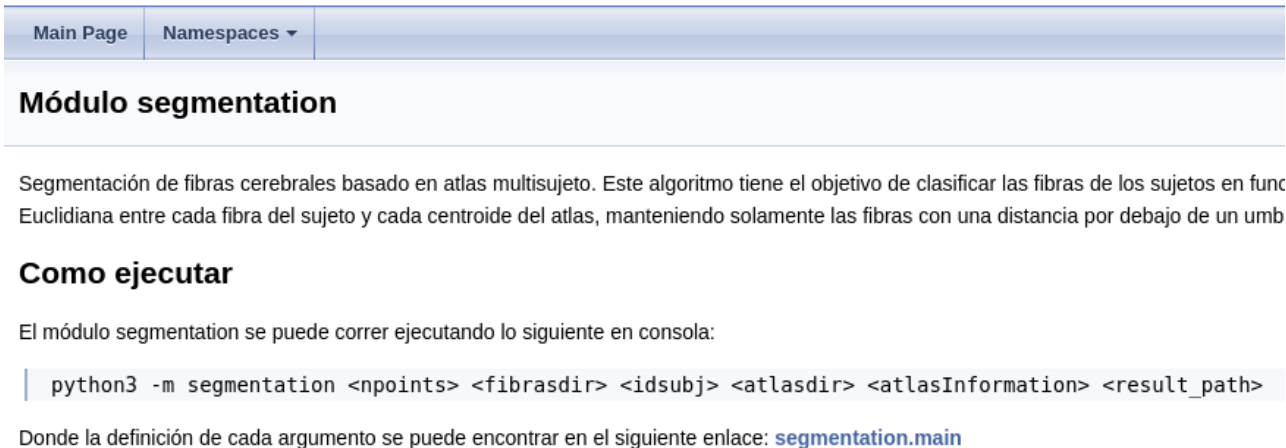


Figura 4.19: Página de inicio de la documentación generada por Doxygen.

4.6. Instalación en Windows

Debido a que el proyecto fue realizado con sistemas operativos como Linux en mente como objetivo principal, se experimentó si los distintos módulos funcionaban en Windows. Para ello, se instaló Python junto con pip y luego, se instaló el paquete desde PyPI mediante pip. Sin embargo, debido a que el sistema operativo objetivo del proyecto era Linux, al ejecutar los códigos se encontraron múltiples falencias como:

- La concatenación manual de rutas, pues debido a que el carácter utilizado en Linux para separar directorios (/) es distinto al de Windows (\), ejecutar los códigos arrojaba errores relacionados a esto. Se solucionó mediante el uso de la función `os.path.join()`, la cual concatena una ruta con el carácter correspondiente para el sistema operativo en que se encuentra.
- El uso de la función `run()`, la cual arroja problemas pues funciona de manera distinta en Windows. Esto se puede resolver mediante el uso de funciones como `os.system()` o utilizando la terminal del Subsistema de Windows para Linux (WSL) [19].
- La falta de compiladores como gcc o g++ para el uso de ciertos programas que requieren compilar estos códigos.

4.7. Resultado Final

Los resultados finales son satisfactorios y es lo que se esperaba lograr. Si bien hay aspectos que se pueden mejorar, como la estructura de la salida de los algoritmos o la compatibilidad entre algunos módulos. El paquete es actualmente funcional y por lo tanto, sus funcionalidades son mucho más accesibles. Por el lado de segmentación, el módulo *segmentation* tuvo muy buenos resultados, con tiempos cortos de ejecución, compatibilidad con *sampling* y la inclusión de los sub-módulos *example* y *help*. De forma similar, el módulo *ffclust* entregó buenos resultados, entre ellos, su compatibilidad con *postprocessing*. En cuanto al módulo *hclust*, si bien no se pudo solucionar el problema del tiempo de ejecución, se pudo sobrepasar la dificultad presentada por la restricción de 60 MB impuesta por PyPI. El módulo *utils*, debido a la especificidad de sus algoritmos y como estos funcionan, posee varias restricciones en cuanto a las entradas que permite, ya sea por extensión o por limitaciones del equipo. Aun así, todos estos sub-módulos que componen *utils*, cumplen con el objetivo que se esperaba. Finalmente, se creó documentación para el archivo principal (`__main__.py`) de cada módulo. Tomando esto en consideración, el resultado final fue lo esperado, una biblioteca en Python que centraliza los códigos de fibras neuronales en un solo paquete, con datos de prueba y documentación que explica como ejecutar los distintos módulos de ésta y que algoritmo representa que cosa.

5. Conclusiones

En este proyecto de memoria de título, se propuso una biblioteca de Python que solucionara los problemas de accesibilidad que afectaban a los códigos desarrollados por el Grupo de Análisis de Imágenes Médicas de la Universidad de Concepción. A pesar de los problemas encontrados durante el desarrollo, se lograron los objetivos satisfactoriamente y el resultado fue el esperado.

De este proyecto se aprendieron técnicas y herramientas para futuros proyectos. Además, este proyecto puede ser usado como base para continuar el trabajo, implementando más funcionalidades y compatibilidad. No solo esto, sino que debido a la estructura del trabajo, como las funcionalidades que lo componen están modularizadas, el modificar un módulo no precisa el generar grandes cambios en los otros, a excepción de un par de funciones.

Este proyecto además, dio a notar la importancia de tomar en consideración la experiencia del usuario durante el desarrollo. De no haberse probado el trabajo desarrollado en una máquina nueva, habrían quedado muchos problemas de compatibilidad y funcionalidad sin resolver, como por ejemplo, garantizar la instalación de las dependencias de la biblioteca, al instalar esta.

Al comparar este proyecto de memoria de título con otros proyectos similares, como DIPY, se puede notar que este trabajo es a una escala bastante menor, pero cumple su propósito de manera efectiva para el estado del arte en el cual se desarrolló.

Referencias

- [1] D. Le Bihan: "Diffusion MRI: what water tells us about the brain." *EMBO Molecular Medicine* 6, 5 (2014), 569–573.
- [2] P.J. Basser, J. Mattiello, and D. LeBihan: "MR Diffusion Tensor Spectroscopy and Imaging." *Biophysical Journal* 66, 1 (1994), 259–267.
- [3] V. Siless, K. Chang, B. Fischl and A. Yendiki: "AnatomyCuts: Hierarchical Clustering of Tractography Streamlines Based on Anatomical Similarity." *NeuroImage* (2016), 184–191.
- [4] P. Guevara, D. Duclap, C. Poupon, L. Marrakchi-Kacem, P. Fillard, D. Le Bihan, M. Leboyer, J. Houenou and J.-F. Mangin: "Automatic fiber bundle segmentation in massive tractography datasets using a multi-subject bundle atlas." *NeuroImage* 61, 4 (2012), 1083–1099.
- [5] A. Vázquez, N. López-López, A. Sánchez, J. Houenoud, C. Poupon, J. Mangin, C. Hernández, and P. Guevara: "FFClust: Fast fiber clustering for large tractography datasets for a detailed study of brain connectivity." *NeuroImage* (2020), –.
- [6] N. Labra, P. Guevara, D. Duclap, J. Houenou, C. Poupon, J.-F. Mangin and M. Figueroa: "Fast Automatic Segmentation of White Matter Streamlines Based on a Multi-Subject Bundle Atlas." *Neuroinform* 15, 71–86 (2017). <https://doi.org/10.1007/s12021-016-9316-7>
- [7] A. Vázquez, N. López-López, N. Labra, M. Figueroa, C. Poupon, J.F. Mangin, C. Hernández, P. Guevara: "Parallel Optimization of Fiber Bundle Segmentation for Massive Tractography Datasets," *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*, 2019, pp. 178-181, doi: 10.1109/ISBI.2019.8759208.
- [8] C. Román, M. Guevara, R. Valenzuela, M. Figueroa, J. Houenou, D. Duclap, C. Poupon, J.-F. Mangin, J.-F. and P. Guevara: "Clustering of whole-brain white matter short association bundles using hardi data." *Frontiers in Neuroinformatics* 11 (2017), 73–73.
- [9] DIPY. (2021) *DIPY - Diffusion Imaging In Python*. Obtenido de <https://dipy.org/>

- [10] L. J. O'Donnell and C.-F. Westin: "Automatic tractography segmentation using a high-dimensional white matter atlas." *IEEE Trans. Med. Imaging* 26, 1562–1575. doi:10.1109/TMI.2007.906785
- [11] T. Kodinariya and P. Makwana: "Review on Determining of Cluster in Kmeans Clustering." *International Journal of Advance Research in Computer Science and Management Studies* 1 (01 2013), 90–95.
- [12] Python Software foundation. (2021) *PyPI · The Python Package Index*. Obtenido de <https://pypi.org>
- [13] The pip developers. (2021) *pip documentation v21.3.1*. Obtenido de <https://pip.pypa.io/en/stable/>
- [14] T. Müller and Manjaro Developers. (2021) *Manjaro homepage*. Obtenido de <https://manjaro.org>
- [15] The Python Packaging Authority. (2021) *setuptools · PyPI*. Obtenido de <https://pypi.org/project/setuptools/>
- [16] The Python Packaging Authority. (2021) *Packaging Python Projects*. Obtenido de <https://packaging.python.org/tutorials/packaging-projects/>
- [17] Python Software Foundation. (2021) *Subprocess management — Python 3.10.0 documentation*. Obtenido de <https://docs.python.org/3/library/subprocess.html>
- [18] D. van Heesch (2021) *Doxygen Homepage* Obtenido de <https://www.doxygen.org/index.html>
- [19] Microsoft. (2021) *Install WSL | Microsoft Docs* Obtenido de <https://docs.microsoft.com/en-us/windows/wsl/install>