Programación de una librería de Python capaz de leer ficheros PDB para representar proteínas en 3D

Cabrelles Muñoz, Aldar
Bioinformática y bioestadística
Área 1 – Programación para la bioinformática

Jiménez García, Brian
Marco Galindo, Maria Jesús

05/06/2018

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial-CompartirIgual 3.0 España de Creative Commons

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Programación de una librería de Python capaz de leer ficheros PDB para representar proteínas en 3D*
Nombre del autor:	*Cabrelles Muñoz, Aldar*
Nombre del consultor/a:	*Jiménez García, Brian*
Nombre del PRA:	*Marco Galindo, Maria Jesús*
Fecha de entrega (mm/aaaa):	06/2018
Titulación:	*Bioinformática y bioestadística*
Área del Trabajo Final:	*Área 1 Programación para la bioinformática*
Idioma del trabajo:	*Español*
Palabras clave	*Python, PDB, viewer*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

Este proyecto tiene como finalidad el uso de librerías de Python para la visualización de estructuras proteicas tridimensionales mediante el procesado de archivos PDB. Para ello, se utilizaron las herramientas ofrecidas por BioPython para el procesado de datos, y las librerías matplotlib y VisPy para la visualización de los datos procesados. El resultado de este trabajo es un programa cuya interfaz gráfica fue creada mediante la librería Tkinter, que permite la selección del archivo a representar, la selección del motor que se utilizará para la visualización y qué tipo de visualización se generará (CPK, según el tipo de aminoácido al que pertenece, dividido en cadenas, y usando DSSP). Para facilitar su uso, también se pueden utilizar los motores por separado en forma de paquetes individuales, MatViewer y VisPyViewer (2D y 3D). Este trabajo se puede utilizar como plataforma para generar nuevas alternativas a los visualizadores existentes, dado que tienen una gran capacidad de personalización, sobre todo gracias a la librería VisPy.

Abstract (in English, 250 words or less):

This project has the objective the use of Python libraries for the visualization of tridimensional protein structures through the processing of PDB archives. For this purpose, the tools offered by BioPython were used for data processing, and the matplotlib and Vispy libraries were used for the visualization of said processed data. The result of this project is a program whose graphic interface was created thanks to the Tkinter library, that allows the archive selection for the representation, the engine selection for the visualization and what visualization type will be generated (CPK, depending on the aminoacidic type, divided by chains and using DSSP). For an easier use, it’s also possible to use the engines separately in the form of individual packages, MatViewer and VisPyViewer (2D and 3D). This project can be used as a platform for generating new alternatives to the already existing visualizers, since they have a great characterization capacity, specially thanks to the VisPy library.

AGRADECIMIENTOS

Gracias a Brian Jiménez por su apoyo incondicional durante el proyecto. Su paciencia y motivación han sido alentadoras tanto para mejorar la calidad del producto y mi formación, como para el crecimiento personal.

Gracias a los desarrolladores David Hoese, por haberme ayudado a entender una gran parte del funcionamiento de las herramientas que he utilizado a lo largo del proyecto, y Luke Campagnola, por haberme enseñado trucos y alternativas que todavía no estaban ni siquiera implementadas al público todavía para mejorar mi trabajo.

Gracias a un gran amigo y compañero que me ayudó a preparar mi sistema para poder realizar este proyecto, Bart Smienk. No solo me ayudó a entender mejor el proceso, y trucos al respecto, sino que siempre me dio apoyo moral a lo largo del proyecto.

Gracias a los profesores que me han formado en la *Universitat Oberta de Catalunya* (UOC) permitiéndome alcanzar lugares en mi formación que de pequeño jamás hubiese imaginado. Gracias a mi tutora del aula Rebeca Sanz, por haberme guiado y ayudado junto al profesorado.

Gracias a los profesores que he tenido a lo largo de mi vida, por los cuales siempre me sentiré afortunado. Sin ellos, no hubiese decidido elegir este camino en el que estoy ahora. Sería injusto que no mencionase en especial a Joana Guijosa por tenerme bajo su ala de protección durante los años más vulnerables de mi vida, Imma Adsuar por encender una chispa en mi curiosidad para el campo químico y bioquímico, y a la doctora Julia Lorenzo, por enseñarme cómo funciona el mundo de la investigación, avivando mi llama por el mundo de la ciencia.

Gracias a todos los compañeros que he tenido estos años, los cuales siempre me inspiran con valor al ver los futuros tan brillantes que han logrado forjar.

Gracias a los amigos que han mostrado una gran paciencia y comprensión a lo largo de mi vida al estar a mi lado, en especial a Raúl Orozco, Joel Martin, Gerard Hernández y Alejandro Araque.

Y, finalmente, gracias a mis padres, José Cabrelles y Julia Muñoz, y hermano, Alberto Sánchez, por haber sido los pilares de mi vida con un apoyo que me ha permitido crecer hasta ser la persona que soy hoy. A todos vosotros, gracias por haber estado siempre ahí.

Índice

1. Introducción 1
1.1 Contexto y justificación del Trabajo 1
¿Qué son las proteínas? ¿Por qué son importantes? 1
Formas para representar las proteínas 2
Bases de datos 5
1.2 Objetivos del Trabajo 6
1.3 Enfoque y método seguido 6
Sistema operativo 7
Python 7
BioPython 8
NumPy 8
Matplotlib 8
Panda3D 8
VisPy 9
Tkinter 9
1.4 Planificación del Trabajo 10
1.5 Breve sumario de productos obtenidos 11
1.6 Breve descripción de los otros capítulos de la memoria 12
2. Protein Viewer 13
2.1 Preparación de datos 13
Datos moleculares 13
PDBParser 14
DSSP 15
Evitar errores 15
2.2 Viewer2d – Matplotlib 16
Preparación plot 16
Visualizaciones 16
CPK 16
Aminoácidos 17
Por cadenas 18
DSSP 18
2.3 Viewer3d – VisPy 19
Shaders 20
Creación de visual 22
Visualizaciones 24
CPK 24
Aminoácidos 24
Por cadenas 25
DSSP 26
Escena y cámara 27
2.4 GUI – Tkinter 28
Selección de archivo 28
Selección de librería 29
Selección de visualización 29
Ejecución 30
2.5 Ejemplo práctico 30
Matplotlib 30
VisPy 34

GUI 37
3. Conclusiones 39
4. Glosario 41
5. Bibliografía 43
6. Anexos 47
6.1 Repositorio 47
6.2 Instalación librerías 47
Python 47
BioPython 47
DSSP 47
NumPy 47
Matplotlib 48
VisPy 48
PyQT4 48
Tkinter 48

Lista de figuras

Ilustración 1 Diferentes formas de visualización proteica. A – Esqueleto; B – Esférico; C – Backbone; D – Ribbon. 4
Ilustración 2 Primer esbozo refinado que representa la idea del GUI diseñado para este programa. 10
Ilustración 3 Diagrama de Gantt que muestra la planificación del proyecto. 11
Ilustración 4 Gráfico explicativo de los pasos que se llevan a cabo con los shaders. Primero se definen los vértices y se aplica el shader `vertex`. A continuación, se genera la base y se rasteriza, y se aplica el shader `fragment`. Después de ser procesado, el framebuffer es la presentación de la imagen de forma física. 20
Ilustración 5 Representación de un grupo de esferas aleatorio en `Testing/spherestest.py`, creado por el usuario Luke Campagnola. 23
Ilustración 6 Visualización CPK del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`. 31
Ilustración 7 Visualización según el tipo de aminoácido del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`. 32
Ilustración 8 Visualización por cadenas del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`. 33
Ilustración 9 Visualización DSSP del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`. 34
Ilustración 10 Visualización CPK del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`. 35
Ilustración 11 Visualización según el tipo de aminoácido del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`. 35
Ilustración 12 Visualización por cadenas del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`. 36
Ilustración 13 Visualización DSSP del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`. 36
Ilustración 14 GUI presente en `pviewer` creada mediante Tkinter para la combinación de los paquetes `MatViewer` y `VisPyViewer`. 37
Ilustración 15 Ventana de selección de archivo tras pulsar el botón `Browse` dentro del GUI presente en `pviewer`. 38
Ilustración 16 Ventana de error que aparece tras pulsar el botón `Run` sin elegir un archivo de tipo PDB dentro del GUI presente en `pviewer`. 38

Tabla 1 Colores asignados según el tipo de aminoácido 13
Tabla 2 Colores asignados según la nomenclatura CPK, y radios atómicos de Van der Waals. 13
Tabla 3 Tabla de colores asignados a las estructuras calculadas mediante DSSP. 14

1. Introducción

1.1 Contexto y justificación del Trabajo

¿Qué son las proteínas? ¿Por qué son importantes?

Las proteínas son las macromoléculas más versátiles fuera de los sistemas vivos, y tienen funciones cruciales en, esencialmente, todos los procesos biológicos. Funcionan como **catalizadores**, transportan y almacenan moléculas como el oxígeno, proveen soporte mecánico y protección inmune, generan movimiento, transmiten impulsos nerviosos, y controlan el crecimiento y el proceso de **diferenciación**.

Estas macromoléculas son capaces de realizar tantas funciones gracias a ciertas propiedades clave:

- Las proteínas son polímeros lineales construidos a base de monómeros llamados aminoácidos. Esta estructura de aminoácidos vinculados se denomina estructura primaria, y dicho vínculo es muy estable. Está formado por unos **enlaces peptídicos** que son capaces de aguantar sin ser **hidrolizados** durante aproximadamente 1000 años, y le da el nombre a la cadena como cadena polipeptídica. Esta cadena, aun así, suele plegarse de forma espontánea en otro tipo de estructuras tridimensionales según la composición aminoacídica, llamadas estructuras secundarias, unidas mediante **puentes de hidrógeno**. Más adelante, las estructuras formadas pueden formar enlaces en largas distancias, que forman otra estructura más avanzada, la estructura terciaria. Las proteínas se asocian con otras y forman lo que se conoce como estructura cuaternaria o complejos.
- Las proteínas tienen una gran variedad de grupos funcionales, como alcoholes, **tioles**, **tioéteres**, **ácidos carboxílicos**, **carboxamidas** y una variedad de grupos básicos. Suelen ser reactivos, y combinados en varias secuencias explican la gran variedad de funciones de las proteínas.
- Las proteínas pueden interaccionar entre ellas mediante otras macromoléculas biológicas que formen complejos que realicen funciones que solas no podrían.
- Las proteínas pueden tener estructuras tanto rígidas como flexibles, permitiéndoles realizar funciones tanto estructurales como conectoras.

Como ya se ha mencionado, los monómeros que forman las proteínas se denominan aminoácidos, y a nivel biológico solo utilizamos un set de 20 distintos. Se agrupan en 4 tipos:

- Aminoácidos hidrofóbicos con grupos apolares.

    - Glicina, Alanina, Prolina, Valina, Leucina, Isoleucina, Metionina, Triptófano, Fenilalanina.
- Aminoácidos polares con grupos neutrales, pero con carga desigualmente distribuida.
    - Serina, Treonina, Tirosina, Asparagina, Glutamina, Cisteína.
- Aminoácidos con carga positiva en pH fisiológico.
    - Lisina, Arginina, Histidina.
- Aminoácidos con carga negativa en pH fisiológico.
    - Aspartato, Glutamato.

La conformación de los enlaces peptídicos se ve restringida debida a la composición de los aminoácidos. El tamaño de cada aminoácido, y los grupos funcionales que lo formen determinarán qué posiciones pueden tener los ángulos de rotación que forman estos enlaces. Esto influirá en cómo de rígida sea la unidad peptídica y en qué estructura secundaria formará en su plegamiento. De esta forma, las estructuras secundarias solo se forman con ciertos requisitos basados en qué aminoácidos formarán parte de estas. Principalmente son (aunque no únicamente):

- Hélices alfa: Esta estructura forma una cadena que gira sobre un eje vertical explotando los puentes de hidrógeno entre CO y NH que se forman entre los aminoácidos. Estas hélices después se suelen agrupar unas sobre otras. Aminoácidos que ramifiquen desestabilizarán la estructura formada.
- Hojas beta: Esta estructura forma un motivo estructural llamado 'cadena beta'. Según cómo se apilen, pueden formar hojas beta paralelas o antiparalelas, y formarán enlaces de hidrógeno como en las hélices alfa. Para que se puedan apilar bien las unas con las otras, se forman unos giros cerrados llamados giros beta.
- También se forman otro tipo de hélices diferentes a las hélices alfa que tienen como función principal un rol estructural, donde se trenzan cadenas peptídicas entre ellas y generan unos complejos fibrilares. Se pueden formar mediante dos o tres cadenas paralelas, formando doble o triple hélices.

Una vez formadas estas estructuras secundarias, se forman otros enlaces y afectan otras fuerzas entre los complejos: **Puentes disulfuro** entre cisteínas, **fuerzas de Van der Waals** e hidrofóbicas. De esta manera, se forman estructuras compactas, solubles en agua con núcleos no polares.[1]

Formas para representar las proteínas

Según la estructura en la que nos centremos, las proteínas se pueden representar de varias formas:

- Estructura primaria: Esta estructura solo implica la representación de la cadena polipeptídica. Normalmente se ve la secuencia en orden mediante su forma resumida de un carácter por aminoácido (a veces en forma resumida de 3 caracteres).
- Estructura secundaria: Para representar la proteína en su estructura secundaria, se suele poner de forma paralela a su

estructura primaria para ayudar a identificar qué aminoácidos pertenecen a que plegamiento de una forma más cómoda. Generalmente, se puede representar mediante códigos de una letra, color o mediante figuras geométricas[2]:

  - Líneas para *random coil*, estructuras flexibles de pliegue aleatorio.
  - Flechas para cadenas beta.
  - Ondulaciones/Zigzag para hélices alfa.

  Hay más estructuras, pero estas son las principales.[3]

- Estructura terciaria/cuaternaria: Aquí ya entramos en simulaciones tridimensionales. Hay diferentes puntos de vista para representar una proteína:
  - Representar la molécula: Como el nombre indica, estamos representando la proteína en sí. Del mismo modo, hay diferentes formas de visualizar la proteína (Ilustración 1Ilustración 1):
    - Esqueleto: Donde vemos todos los átomos enlazados en la proteína.
    - Esférico: Donde vemos los aminoácidos en forma de esferas con sus volúmenes permitiendo ver cómo interactúan entre ellos.
    - *Backbone*: Donde vemos principalmente la cadena principal de los enlaces peptídicos. Se pierde mucho detalle, pero permite ver mejor la estructura interna.
    - *Ribbon*: Es una representación donde se pierde la visualización realista, pero que permite identificar qué tipo de plegamiento se lleva a cabo mucho más fácilmente mediante el uso de símbolos como en la estructura secundaria.[2]
  - Representar la superficie: Parecido a la forma esférica, se puede representar la superficie de la proteína de una forma menos precisa, pero que ayuda a identificar qué forma tiene la proteína de una forma más general.
  - Otras representaciones: De la misma forma que la superficie, también se pueden representar las fuerzas que generan las proteínas, como por ejemplo las de van der Waals.[4]

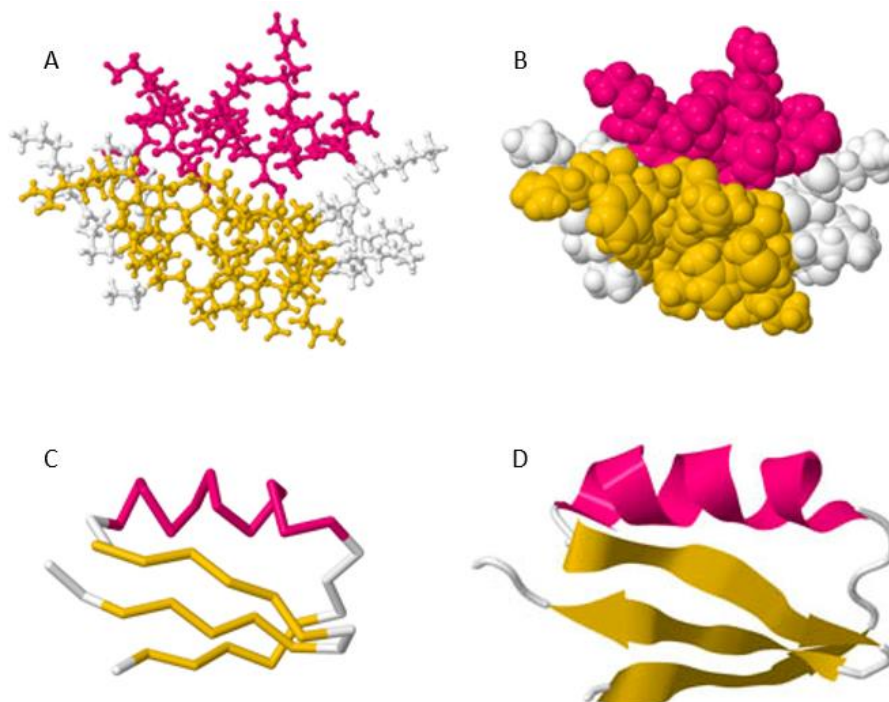

Ilustración 1 Diferentes formas de visualización proteica.[2] A – Esqueleto; B – Esférico; C – *Backbone*; D – *Ribbon*.

- Herramientas para representar las proteínas

A lo largo del tiempo, se han creado muchas herramientas para predecir estructuras proteicas, representarlas y visualizarlas. Muchas de ellas se encuentran agrupadas en ExPASy[5]. Nos centramos en algunas las herramientas de visualización (no predicción) que se encuentran en esta página.

- Swiss-PdbViewer[6]: Permite analizar la estructura de diferentes proteínas a la vez de una forma tridimensional, utilizando archivos PDB.
- SwissDock[7]: Permite mostrar cómo interactúan pequeñas moléculas con las zonas activas de una proteína.
- SwissParam[8]: Provee al usuario con una topología y parámetros de moléculas orgánicas pequeñas.
- Jmol[9]: Miniaplicación en lenguaje java interactiva para navegador web que permite visualizar estructuras químicas en tres dimensiones.
- MarvinSpace[10]: Herramienta de visualización en 3D de alto rendimiento, permitiendo visualización y manipulación.
- PyMOL[11]: Paquete de software que permite **renderizar** y animar estructuras tridimensionales.
- RasMol[12]: Aplicación para la visualización molecular de uso sencillo y rápido.
- SRS 3D[13]: Banco de trabajo para navegador web que incluye tanto la representación tridimensional de la proteína buscada como información de su función y contexto.

Obviamente, tanto dentro como fuera de esta página se encuentran una infinidad más de programas útiles para la visualización. Otros ejemplos son:

- Cn3D[14]: Procesa en el navegador imágenes en 3D de las moléculas, disponible por el NCBI.
- Chimera[15]: Permite analizar moléculas y distorsionarlas para ver los cambios estructurales que esto genera.

Ahora bien, Python también se encuentra presente en este campo de la bioinformática. Hay ejemplos de uso de matplotlib para representación de proteínas de forma básica[16], y librerías creadas para el análisis estructural de proteínas, como ProDy[17]. Otro ejemplo de librería es MDAnalysis, que aprovecha las coordenadas que proveen diferentes tipos de archivos para la creación de modelos tridimensionales, permitiendo la selección de átomos dentro de la estructura creada.[18]

Aun así, no he encontrado ejemplos del uso de Panda3D para esta función, menos el trabajo "*Dock It*" de Brian Jiménez García.[19]

VisPy es otra librería que permite representación en 3D, esta vez basada en OpenGL[20]. Esta librería en concreto ya se ha utilizado para crear visualizadores de proteínas, como por ejemplo *pyball*, por Bosco Ho[21]. Esta librería en concreto es mucho más ligera que Panda3D, aunque pueda requerir una mayor preparación debido al uso de OpenGL.

Bases de datos

Los investigadores requieren organizar, procesar y estructurar los datos para poder difundirlos y compartirlos a nivel público. Para ello, se utilizan las bases de datos, que se pueden diferenciar principalmente en:

- Bases de datos primarias. Estas bases de datos contienen la información biomolecular en su forma original, directamente presentada por los investigadores.
  Las combinaciones de bases de datos primarias se denominan bases de datos compuestas.
- Bases de datos secundarias. Estas bases de datos contienen información derivada del análisis de las fuentes primarias. Son creadas manualmente.[22]

A nivel proteico, una de las bases de datos más importantes es el archivo *Protein Data Bank* (PDB). Es la única base de datos (primaria) desde 1971 que ha funcionado como el repositorio de información para las estructuras tridimensionales de proteínas, ácidos nucleicos y ensamblajes complejos.[23] Esta base de datos ayuda tanto a estudiantes como a investigadores entender todos los aspectos de biomedicina y agricultura, desde la síntesis proteica hasta la salud y enfermedad.[24]

A nivel de base de datos secundaria proteica, DSSP es un programa diseñado por Wolfgang Kabsch y Chris Sander para estandarizar la asignación de estructuras secundarias. DSSP es una base de datos de asignación de estructuras secundarias (y más) para todas las proteínas

que se encuentran en el PDB. Es importante mencionar que DSSP no predice las estructuras secundarias, las calcula a partir de las coordenadas tridimensionales de los aminoácidos.[25]–[27]

1.2 Objetivos del Trabajo

Los objetivos generales de este trabajo son la mayor formación en Python para la Programación Orientada a Objetos, para la visualización de estructuras 3D mediante las librerías matplotlib y *Panda3D* o VisPy, y para la creación de una Interfaz de Usuario Gráfica (GUI) mediante la librería Tkinter[28].

Los objetivos específicos de este trabajo son:

1. Lectura de *Think Python*[29] para una mayor formación en *Python* e introducción a Tkinter.
2. Lectura de documentación de BioPython para una mayor formación en la librería.
3. Lectura de documentación de matplotlib para una mayor formación en la librería.
4. Lectura de documentación de Panda3D para una mayor formación en la librería.
5. Lectura de documentación de VisPy para la elección y mayor formación en la librería.
6. Preparación de la visualización de dos estructuras mediante matplotlib.
   a. Representación con todos los átomos.
      i. **CPK**.
      ii. Según el tipo de aminoácido.
   b. Representación de la cadena polipeptídica Cα-N.
      i. Diferenciando cadenas.
      ii. Utilizando el programa DSSP.
7. Preparación de la visualización de dos estructuras mediante VisPy.
   a. Representación con todos los átomos.
      i. CPK.
      ii. Según el tipo de aminoácido.
   b. Representación de la cadena polipeptídica Cα-N.
      i. Diferenciando cadenas.
      ii. Utilizando el programa DSSP.
8. Creación de una GUI mediante Tkinter.

1.3 Enfoque y método seguido

Para realizar este proyecto, se decidió trabajar en Xubuntu[30] (Ubuntu + Xfce) para programar en un entorno Python. Se propuso el uso de archivos PDB para obtener la información necesaria para la representación proteica, y para su extracción y organización se usó la librería BioPython. Para organizar las diferentes matrices de datos, se utilizó la librería NumPy, y para la representación visual se propuso el uso de Matplotlib y una librería alternativa, a elegir entre Panda3D y VisPy. Finalmente, para la creación de una GUI, se utilizó la librería Tkinter.

El ordenador utilizado tiene una placa base *GA-H61M-DS2* de fabricante *Gigabyte Technology Co.*, un procesador *Intel® Core™ i5-2300*, y 8192MB de memoria RAM. La tarjeta gráfica que tiene el ordenador utilizado es *NVIDIA GeForce GTX 960* con una memoria VRAM de 4053 MB.

Sistema operativo

Xubuntu es un sistema operativo que combina la versatilidad de Ubuntu[31] y la ligereza del entorno Xfce.[32]

Ubuntu es un software de código abierto que permite la instalación de módulos y programas de formas rápidas y sencillas, aprovechando el uso de repositorios tanto oficiales como no oficiales en nubes online.[31]

Xfce es un entorno de escritorio ligero para sistemas tipo UNIX, cuyo objetivo es ser rápido y usar pocos recursos del sistema, sin dejar de ser visualmente atractivo y fácil de usar.[32]

Aprovechando las características de este software, Ubuntu siendo usado también a lo largo del Máster en Bioinformática y Bioestadística, decidí aprovechar Xubuntu como una alternativa más ligera para su uso prolongado.

Python

Python es un lenguaje de programación creado en los años 1990s por Guido van Rossum, y ha seguido evolucionando desde entonces de una forma consistente.[33] Es muy sencillo de entender, permitiendo el aprendizaje tanto para programadores experimentados como para novatos en el campo. Su instalación es sencilla, y a veces preinstalada en algunos sistemas operativos.[34]

Es además versátil, permitiendo el uso de librerías de acceso público de campos completamente distintos[35]:

- Desarrollo de Webs e Internet.
- Científico y numérico.
- Educación.
- GUI's de escritorio.
- Desarrollo de software.
- Aplicaciones para negocios.

Python utiliza dos versiones distintas principalmente, la 2.x y 3.x. Hoy en día, a pesar de que Python sigue evolucionando, se sigue utilizando Python 2.x en muchos entornos para evitar problemas de incompatibilidades de librerías.[36]

Por estas razones, este proyecto utilizará este lenguaje.

BioPython

BioPython es un conjunto de herramientas de acceso gratuito en Python creado por una asociación internacional de desarrolladores, con uso orientado a la biología computacional.[37]

Biopython tiene un módulo que se centra en el trabajo de estructuras cristalinas de macromoléculas biológicas: `Bio.PDB`. Es una herramienta capaz de acceder a los datos atómicos de una forma completa y consistente utilizando archivos PDB.[38]

En concreto, este módulo tiene dos herramientas que serán de gran ayuda a lo largo del proyecto: `Bio.PDB.PDBParser` y `Bio.PDB.DSSP`.

La primera herramienta permite el acceso a los datos atómicos del archivo PDB, en una jerarquía de tipo estructura →modelo →cadena →residuos →átomos, ofreciendo una gama de datos muy amplia en cada escalón.[38]

La segunda herramienta permite el uso de la base de datos DSSP, que calcula la estructura secundaria de cada aminoácido según sus coordenadas en el modelo tridimensional.[39]

Con esta librería, se evita la necesidad de crear un *parser* que nos transforme el código a componentes funcionales, de forma autónoma, y permite añadir nuevas características a las visualizaciones.

NumPy

Numpy es un paquete fundamental para la computación científica en Python. Es una gran herramienta tanto para la creación de objetos de tipo matriz y diferentes funciones que incluyen, entre otras, crear contenedores multidimensionales de datos de tipo genérico. Esto permite la integración de diferentes bases de datos.[40]

Por esta razón, NumPy se utiliza a lo largo del proyecto para la organización de los diferentes datos analizados y obtenidos.

Matplotlib

Matplotlib es una librería de ***plotting*** (creación de gráficos) en 2D para Python que permite la creación de ilustraciones y figuras con una amplia libertad de formatos para diferentes plataformas. Es de fácil acceso y una de sus funciones posibles es la creación de *plots* en ejes tridimensionales.[41]

De esta forma, una de las visualizaciones se realizará con esta librería.

Panda3D

Panda3D es un motor de videojuegos que permite el desarrollo y renderización para programas Python y **C++**. Esta librería se creó con el objetivo de ser fuerte, rápida, completa (con herramientas esenciales) y

que con tolerancia de errores (evita que el programa se rompa y facilita la localización de los errores).[42]

Aunque tiene herramientas muy útiles que permiten crear modelos interactivos, con diferentes propiedades, a nivel de crear modelos atómicos puede causar problemas: cada átomo se renderiza de forma individual, con sus características, y esto ralentiza mucho el proceso de carga, sobre todo si se quieren crear nuevas funciones encima. Esta es la razón principal por la que no se utilizó esta librería.

VisPy

VisPy es una librería de *Python* para la visualización científica interactiva en 2D y 3D que es diseñada para ser rápida, escalable y fácil de utilizar. Aprovecha la potencia del **GPU** en lugar del **CPU** mediante la librería de OpenGL. De esta forma, se aligera la carga del ordenador y permite personalizar incluso más la visualización.[43]

Como requisito adicional, requiere la instalación de un ***back-end*** gráfico, pero su instalación es sencilla y hay diferentes posibilidades. En este proyecto, se utiliza PyQt4. PyQt4 es la implementación de Qt para Python, que es un conjunto de librerías C++ y de herramientas de desarrollo.[44]

Aunque se requiera conocimientos de OpenGL, algo que personalmente no tengo muy desarrollados, hay muchos códigos de acceso público desarrollado por la comunidad, de los cuales se aprovecharán ciertas herramientas.

Tkinter

Tkinter es el paquete de GUI estándar de *Python* más comúnmente utilizado.[45]

El primer concepto que pensé para crear una GUI tenía una estructura muy sencilla que esta librería podría aprovechar, con una forma de seleccionar el archivo, librería y tipo de visualización por separado, previo a un botón para la ejecución final (Ilustración 2).

Ilustración 2 Primer esbozo refinado que representa la idea del GUI diseñado para este programa.

Es fácil de aprender y utilizar, y da la libertad de diseño suficiente para crear un programa sencillo que unifique las librerías. Por estas razones, se utilizará esta librería a lo largo del proyecto.

1.4 Planificación del Trabajo

Para desarrollar este programa, se decidió separar el proyecto en 2 bloques principales:

1. Formación:
    a. Preparación para Python y su programación orientada a objetos.
    b. Formación para el uso de matplotlib.
    c. Selección entre Panda3D y VisPy.
    d. Formación para el uso de la librería seleccionada, en este caso VisPy.
2. Programación:
    a. Programación matplotlib
    b. Programación VisPy
    c. Programación GUI con Tkinter

De esta forma, separé los contenidos de cada PEC a partir del segundo bloque:

1. Para la primera parte se finalizó la formación y el bloque de matplotlib
2. Para la segunda parte se prosiguió con la formación de VisPy, se finalizó su bloque y se preparó un archivo Python con una GUI hecha en Tkinter.

La planificación del proyecto se puede ver en el diagrama de Gantt de la Ilustración 3.

Febrero | Marzo
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 | 1 2 3
Planificación
Lectura Think Python (POO)
Formación BioPDB

Marzo
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Lectura Think Python (POO)
Formación BioPDB
Formación matplotlib
Elegir entre Panda3D y VisPy
Formación librería 3D

Abril | M
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | 1
Programación matplotlib I
Programación matplotlib II
Programación 3D I
Programación 3D II

Mayo
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Programación 3D II
Refinar/Más desarrollo (?)
Redacción memoria

Junio | J
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | 31
Redacción memoria
Preparar presentación
Defensa pública

Ilustración 3 Diagrama de Gantt que muestra la planificación del proyecto.

1.5 Breve sumario de productos obtenidos

Los productos obtenidos con este proyecto son:

- Paquete que permite visualizar proteínas en 3D basado en matplotlib.
- Paquete que permite visualizar proteínas en 3D basado en VisPy.
- Archivo Python que permite ejecutar un GUI basado en Tkinter usando los dos paquetes anteriores. Este GUI permite seleccionar el archivo PDB, la librería que utilizar y qué visualización realizar.

Ambos paquetes permiten el control de la cámara con el ratón y permiten la visualización de archivos PDB de 4 formas diferentes:

1. CPK.
2. Según el tipo de aminoácido (apolar/apolar con o sin cargas).
3. Diferenciando cadenas polipeptídicas.
4. Aprovechando el programa DSSP.

Dentro del anexo, el enlace al repositorio de GitHub se puede encontrar en 6.1 Repositorio, y la instalación de los módulos y librerías necesarias se puede encontrar en 6.2 Instalación librerías.

1.6 Breve descripción de los otros capítulos de la memoria

En el siguiente capítulo se explicará en un poco más de profundidad el uso de las diferentes librerías dentro del software creado, no enfocado al contexto, sino centrado en su programación.

- En el primer apartado se explicará cómo se procesan los datos del archivo PDB de forma común para ambos tipos de visualización.
- Su siguiente apartado explicará en concreto el uso de matplotlib para las diferentes visualizaciones.
- El penúltimo apartado explicará en concreto el uso de VisPy para las diferentes visualizaciones.
- Finalmente, el último apartado explicará en concreto el uso de Tkinter para la creación de una GUI.

2. Protein Viewer

Los paquetes creados en este proyecto tienen una estructura común:

- Procesado del archivo PDB mediante PDBParser y DSSP.
- Organización de los datos según color, radio, coordenadas y en algunos casos cadenas.
- Preparación del contexto de las librerías de visualización (matplotlib y VisPy).
- Aplicación de los datos organizados por matrices dentro de los contextos preparados.

Estos pasos se verán delimitados y variados según el tipo de visualización que se esté exigiendo inicialmente.

Los paquetes creados son `Viewer2d` con la función `MatViewer` que permite la visualización de estructuras mediante matplotlib, y `Viewer3d` con la función `VisPyViewer` que permite la visualización de estructuras mediante VisPy.

2.1 Preparación de datos

A lo largo del trabajo hay pasos y paquetes que se repiten dado a que son de preparación: se obtienen los datos y se organizan para su posterior desglosado y uso en las librerías de visualización concretas, además de asegurar que se seleccionan las opciones correctamente.

Datos moleculares

En ambos paquetes `Viewer2d` y `Viewer3d` hay un archivo llamado `molecular_data.py`. Como su nombre indica, este archivo contiene diccionarios y funciones esenciales para facilitar la organización de los datos.

Los datos se encuentran resumidos en Tabla 1, Tabla 2 y Tabla 3.

Tabla 1 Colores asignados según el tipo de aminoácido.[1]

Tipo	Color	Residuos
Apolar	Gris	Ala, Val, Leu, Ile, Met, Pro, Phe, Trp, Gly
Polar sin carga	Morado	Ser, Thr, Gln, Asn, Tyr, Cys
Polar negativo	Añil	Asp, Glu
Polar positivo	Rojo oscuro	Lys, Arg, His

Tabla 2 Colores asignados según la nomenclatura CPK[46], y radios atómicos de Van der Waals.[47]

Elemento	Color	Radio (Angstroms)
H	Blanco	1.2
C (y sus variantes)	Negro	1.7
N	Azul	1.55
O	Rojo	1.52

F	Verde	1.47
Cl	Verde	1.75
Br	Marrón	1.85
I	Violeta oscuro	1.98
He	Turquesa	1.4
Ne	Turquesa	1.54
Ar	Turquesa	1.88
Xe	Turquesa	2.16
Kr	Turquesa	2.02
P	Naranja	1.8
S	Amarillo	1.8
B	Salmón	1.82
Li	Lila	1.82
Na	Lila	2.27
K	Lila	2.75
Rb	Lila	3.03
Cs	Lila	3.43
Be	Verde oscuro	1.53
Mg	Verde oscuro	1.73
Ca	Verde oscuro	2.31
Sr	Verde oscuro	2.49
Ba	Verde oscuro	2.68
Ra	Verde oscuro	2.83
Ti	Gris	*1.5**
Fe	Naranja oscuro	*1.5**
Otros	Rosa	*1.5**

**Valores arbitrarios*

Tabla 3 Tabla de colores asignados a las estructuras calculadas mediante DSSP[39].

Abreviación	Estructura secundaria	Color
H	Hélice alfa (4-12)	Rojo
B	Residuo aislado de puente beta	Marrón
E	Hebra	Amarillo
G	Hélice 3-10	Lila
I	Hélice Pi	Rosa
T	Giro	Turquesa
S	Pliegue	Verde
-	Nada	Gris

PDBParser

PDBParser es una herramienta dentro de BioPDB que permite procesar un archivo PDB para utilizar sus datos atómicos en profundidad.[38] Su uso es muy sencillo, y dentro de la clase objeto se presenta así:

```
from Bio.PDB import PDBParser

#[...]
        self.parser = PDBParser(QUIET=True,PERMISSIVE=True)
        self.structure =
self.parser.get_structure('model',pdbdata)
```

De esta forma, dentro de `self.structure` tenemos almacenados los datos a utilizar más adelante.

DSSP

DSSP es otra herramienta dentro de BioPDB que permite calcular la estructura secundaria que tendrá un aminoácido según su localización tridimensional.[39]

Dentro de la clase objeto programada, obtener los valores DSSP es muy sencillo:

```
from Bio.PDB import PDBParser,DSSP


#[...]
        self.model = self.structure[0]
        self.dssp = DSSP(self.model, pdbdata)
```

Dentro de `self.dssp` tenemos un diccionario están guardados los datos DSSP, y en su segunda columna tenemos guardados todas las abreviaciones de la estructura secundaria. De esta manera, si queremos esta lista, podemos hacer lo siguiente:

```
struct3 = [dssp[key][2] for key in list(dssp.keys())]
residues = [residue for residue in structure.get_residues() if
residue.get_resname() in resdict.keys()]
```

En `struct3` tendremos guardados todas las abreviaciones de DSSP, y en `residues` tendremos el listado de todos los residuos que se encuentren dentro de las listas de residuos de `molecular_data.py`. Hacemos esto porque el *parser* también puede añadir H2O como residuos, y eso no nos interesa.

Evitar errores

Para evitar que el programa nos cause ningún problema al seleccionar el modo de representación, podemos añadir una línea de código muy sencilla. Hay que crear una lista con los modos posibles, y algo que bloquee el uso de modos que no estén en nuestra lista:

```
    visualization_modes = ['cpk','backbone','aminoacid','dssp']

#[...]

        if mode not in
[MatViewer|VisPyViewer].visualization_modes:
            raise Exception('Not recognized visualization mode
%s' % mode)
        self.mode = mode
```

Hay que decir que el `__init__` de nuestra clase objeto contiene tanto `pdbdata` (el archivo PDB) como el modo a representar:

```
    def __init__(self, pdbdata, mode='cpk'):
        #[...]
```

2.2 Viewer2d – *Matplotlib*

El paquete creado en `Viewer2d` consta del objeto clase `MatViewer`. Este paquete nos permite visualizar las estructuras del archivo PDB aprovechando la librería matplotlib. El problema que tiene matplotlib es que las creaciones de las nubes de puntos se hacen por color, y como resultado tendremos un conjunto de nubes de puntos dentro del mismo eje. Esto ralentiza mucho la visualización, y limita su organización.

Preparación *plot*

Independientemente de la visualización, al principio de cada función tendremos que añadir la figura y los ejes tridimensionales, y se finaliza quitando los ejes y mostrando el *plot*.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#[...]
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')

#[...]

        ax.axis("off")
        plt.show()
```

Visualizaciones

Todas las visualizaciones utilizan un *loop* (bucle) que cambiará según el color.

CPK

Según el color y el tipo de átomo en los diccionarios creados en `molecular_data.py`, podemos crear un *loop* donde se almacenan las coordenadas. Además, nos aseguramos de que solo se creen los *plots* si hay al menos un elemento en cada lista.

```
#[...]

        for atom_type, color in colors.iteritems():
            atoms = [atom for atom in self.structure.get_atoms()
if atom.get_id() == atom_type]
            coordinates = [atom.coord for atom in atoms]
            if not len(atoms)==0:
                x, y, z=zip(*coordinates)
                ax.scatter(x, y, z, c=color, marker='o')

#[...]
```

Para poder representar los átomos que no se encuentran en el diccionario de colores, restamos la lista de átomos total menos la de átomos que se encuentran en el diccionario.

```
#[...]

        atoms_1 = [atom for atom in self.structure.get_atoms()]
        atoms_2 = [atom for atom in self.structure.get_atoms()
if atom.get_id() in colors.keys()]
        atoms_pink = list(set(atoms_1)-set(atoms_2))

        coordinates_pink = [atom.coord for atom in atoms_pink]
        xp, yp, zp = zip(*coordinates_pink)
        ax.scatter(xp, yp, zp, c='pink', marker='o')

#[...]
```

Aminoácidos

Esta representación es bastante más complicada. De la misma forma que en CPK, creamos un *loop* según el residuo y color. Tendremos que unir las matrices para crear una sola matriz que contenga todos los puntos para un solo *plot*, por cada *plot*.

```
#[...]

        for resname, residuetype in resdict.iteritems():
            residues = [residue for residue in
self.structure.get_residues() if residue.get_resname() ==
resname]
            rescoord = []
            color = colorrgba(residuetype)

            for residue in residues:
                atoms = [atom for atom in residue.get_atoms()]
                coordinates = [atom.coord for atom in atoms]
                rescoord.append(np.array(coordinates))

            if len(rescoord)>1:
                rescoord = np.concatenate(rescoord)

            if not len(residues)==0:
                x, y, z =zip(*rescoord)
                ax.scatter(x, y, z, c=color, marker='o')

#[...]
```

Después, de forma extra, añadimos los átomos de los residuos que no pertenecen al diccionario (sin contar agua).

```
#[...]

        residues_1 = [residue for residue in
self.structure.get_residues() if residue.get_resname() != 'HOH']
```

```
        residues_2 = [residue for residue in
self.structure.get_residues() if residue.get_resname() in
resdict.keys()]

        residues_pink = list(set(residues_1)-set(residues_2))
        rescoordpink = []

        for residue in residues_pink:
            atomspink = [atom for atom in residue.get_atoms()]
            coordinatespink = [atom.coord for atom in atomspink]
            rescoordpink.append(np.array(coordinatespink))

        if len(rescoordpink)>1:
            rescoordpink = np.concatenate(rescoordpink)

        xp, yp, zp = zip(*rescoordpink)
        ax.scatter(xp, yp, zp, c='pink', marker='o')
#[...]
```

Por cadenas

Esta es la representación más sencilla. Aprovechando la función de NumPy `random.rand()`, podemos crear una matriz de color aleatoria, randomizando los colores de cada cadena. Así, podemos crear una línea y una nube de puntos que en conjunto se conecten. En el momento de elegir átomos, podemos elegir cuales necesitamos, y en este caso podemos hacer la cadena polipeptídica mediante Cα-N. Si por alguna razón el uso de nitrógenos generase alguna aberración, se podría corregir muy fácilmente a Cα-Cα eliminando la sección que elige los átomos N.

```
#[...]

        for chain in self.structure.get_chains():
            can_atoms = [atom for atom in chain.get_atoms() if
atom.get_name() == 'CA' or atom.get_name() == 'N']
            can_coordinates = [atom.coord for atom in can_atoms]
            x,y,z=zip(*can_coordinates)
            ccolor = np.random.rand(3,1)

            ax.plot(x, y, z, c=ccolor, linewidth=2)
            ax.scatter(x, y, z, c=ccolor, marker='o')

#[...]
```

DSSP

Esta es la representación más complicada. La parte sencilla es la creación de líneas que unan las nubes de puntos, ya que es exactamente la misma función que la encontrada en la sección anterior, pero eliminando:

```
ax.scatter(x, y, z, c=ccolor, marker='o')
```

La forma en la que se obtiene el listado DSSP y de residuos está en la sección anterior DSSP dentro de 2.1 Preparación de datos. Se realiza un paso extra, que es la creación de una variable `respred` con el conjunto de residuo y DSSP.

```
respred = zip(struct3,residues)
```

Con este listado podemos crear un *loop* junto a los diccionarios que se encuentran en `molecular_data.py`. Lo que conseguimos con esto es que por cada elemento DSSP que coincida con el diccionario, se añadirá al grupo del color que tiene asignado este diccionario. Una vez se separe cada residuo por color, podemos elegir sus átomos y representarlos en una nube de puntos.

```
#[...]

        for prediction, color in colorsDSSP.iteritems():
            residuesp = [residue[1] for residue in respred if
residue[0] == prediction]

            predcoord_can = []
            for residue in residuesp:
                atomsp = [atom for atom in residue.get_atoms()
if atom.get_name() == 'CA' or atom.get_name() == 'N']
                coordinatesp = [atom.coord for atom in atomsp]
                predcoord_can.append(np.array(coordinatesp))

            if len(predcoord_can)>1:
                predcoord_can = np.concatenate(predcoord_can)

            if not len(residuesp)==0:
                x, y, z = zip(*predcoord_can)
                ax.scatter(x, y, z, c=color, marker='o')

#[...]
```

Como en el caso de los residuos, es importante unir todos los átomos del mismo color en una sola matriz en caso de que haya más de un elemento por elemento DSSP.

2.3 Viewer3d – VisPy

El paquete creado en `Viewer3d` consta del objeto clase `VisPyViewer`. Este paquete nos permite visualizar las estructuras del archivo PDB aprovechando la librería VisPy. Esta librería permite la creación de modelos con la extensa capacidad de OpenGL/GLSL.[43]

OpenGL es una Interfaz de Programación de Aplicaciones (API) 2D y 3D creada en 1992. Esta API es actualmente la más utilizada y extensa en industrias que desarrollan aplicaciones gráficas.[48]

Al principio, OpenGL tenía una estructura ***pipeline*** (línea de trabajo) fija, donde la transformación de los datos es lineal. Esto quiere decir que primero se tiene que definir un sistema de coordenadas de los píxeles y su **rasterización** a pixeles con color. A lo largo del tiempo y a medida que las interfaces gráficas han ido mejorando, este tipo de *pipelines*

empezaron a complicar su uso: hay que tener en cuenta la iluminación, texturas, reflejos y más detalles, aumentando el número de operaciones.[49]

Más adelante, y para añadir flexibilidad a este proceso, se dio la posibilidad de customizar el *pipeline* de renderizado a los programadores. De esta forma, se permitió usar pequeños programas con un lenguaje llamado *shading language* (en OpenGL, el lenguaje de *shaders* es llamado GLSL). Estos programas, llamados *shaders*, se ejecutan independientemente en paralelo a lo largo del código principal, y transforma los vértices y píxeles mediante sus propios algoritmos. Los *shaders* más utilizados son, como se verá más adelante, `vertex` (ejecutado por cada vértice) y `fragment` (ejecutado por cada pixel). Los *shaders* son extremadamente potentes, ya que dan al programador control total de la tarjeta gráfica para que hagan un renderizado a tiempo real.[49]

VisPy tiene a su disposición una gran cantidad de visuales[50] que permiten como base crear escenas (interactivas o no) a medida para el usuario.[51] En caso de que el usuario tenga conocimientos de OpenGL, puede aprovechar `vispy.gloo`, que crea un programa a partir de los *shaders* creados por el propio usuario.[52] También se pueden aprovechar estos *shaders* para crear visuales personalizados aplicables a una escena interactiva.[53]

Shaders

El código GLSL de `vispy.gloo` (y la creación de visuales más adelante) se construye a partir de dos *shaders*: el `vertex` y el `fragment`. El resumen del proceso se puede ver en la Ilustración 4.

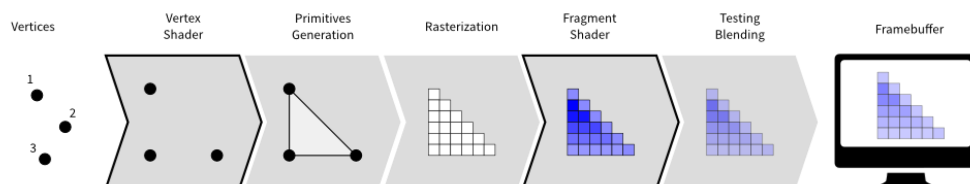

Ilustración 4 Gráfico explicativo de los pasos que se llevan a cabo con los *shaders*. Primero se definen los vértices y se aplica el *shader* `vertex`. A continuación, se genera la base y se rasteriza, y se aplica el *shader* `fragment`. Después de ser procesado, el *framebuffer* es la presentación de la imagen de forma física.[54]

Para programar un *shader*, necesitamos dos partes:

1. `vertex`: Es la parte del código que proporciona las posiciones de un vértice. De una forma resumida, es 'qué' píxeles vamos a renderizar.
2. `fragment`: Es la parte del código que proporciona las características visuales de los vértices. De una forma resumida, es el 'cómo' vamos a renderizar los píxeles.

Los *shaders* pueden tener inputs y outputs de datos en 3 formatos diferentes:

- `attribute`: Los atributos solo están disponibles en los `vertex`, y solo sirven para lectura de datos.
- `uniform`: Son valores que no cambian a lo largo del renderizado, están disponibles en `vertex` y `fragment`, y solo sirven para lectura de datos.
- `varying`: Son valores que se utilizan para pasar datos de un `vertex` a un `fragment`. En `vertex` son elementos que pueden leerse y sobrescribirse, y se reutilizan en `fragment`, pero aquí solo sirven para lectura de datos.

Hay diferentes elementos construidos para cada tipo de formato del *shader*:

- *Atributes*
  - `gl_Vertex`: vector 4D que representa la posición del `vertex`.
  - `gl_Normal`: vector 3D que representa la normal del `vertex`.
  - `gl_Color`: vector 4D que representa el color del `vertex`.
  - `gl_MultiTexCoordX`: vector 4D que representa la coordenada de textura de la unidad de textura X.
- *Uniforms*
  - `gl_ModelViewMatrix`: Matriz 4x4 que representa la matriz de vista-modelo.
  - `gl_ModelViewProjectionMatrix`: Matriz 4x4 que representa la matriz de modelo-vista-proyección.
  - `gl_NormalMatrix`: Matriz 3x3 que representa el transpuesto inverso de la matriz de modelo-vista. Esta matriz se utiliza para transformación normal.
- *Varyings*
  - `gl_FrontColor`: vector 4D que representa el color frontal.
  - `gl_BackColor`: vector 4D que representa el color trasero.
  - `gl_TexCoord[X]`: vector 4D que representa la textura de la coordenada X.

GLSL tiene tipos de outputs predefinidos para el *shader* especialmente importantes en `vispy.gloo`:

- `gl_Position`: vector 4D que representa la posición del vértice procesado, solo en el `vertex`.
- `gl_FragColor`: vector 4D que representa el color final que se escribe en el *buffer* del *frame*, solo en el `fragment`.
- `gl_FragDepth`: elemento de tipo `float` (decimal) que representa la profundidad en la que se escribe el buffer de profundidad, solo disponible en `fragment`.[54]

Con todo esto en mente, ya se puede crear un programa capaz de mostrar las proteínas mostrando cada átomo. Un ejemplo de esto es el ejemplo número 30 de la galería de *VisPy*: `molecular_viewer.py`.[55]

Aun así, la creación de un visual para su uso en una escena facilitaría su sistematización y uso con otros visuales.

Creación de visual

Para crear un visual, hay que programar los *shaders* como si fuesen para `vispy.gloo`, pero a nivel de plantilla que se pueda rellenar con funciones Python personalizables.[53]

Es importante tener en cuenta que cada vez que se dibuja un visual, se le da una instancia de `TransformSystem` que transfiere la información sobre el tamaño de los píxeles lógicos y físicos relativos al visual, en cuatro sistemas de coordenadas diferentes:

`Visual` → `Document` → `Framebuffer` → `Render`

Esta diferenciación es importante porque el uso de las transformaciones predeterminadas en VisPy pueden causar problemas al no estar correctamente especificadas.[56]

Tras alcanzar a la comunidad de VisPy, conocí a los desarrolladores que me mostraron visuales que todavía no estaban en el repositorio y me permitieron su uso, en concreto el usuario Luke Campagnola[57].

Este desarrollador me mostró un ejemplo de visual que todavía no está en el repositorio (pronto lo estará), y se nos dio permiso para utilizarlo. El visual creado es capaz de crear esferas con profundidad con una mayor calidad que con `molecular_viewer.py`. Se aprovechó y reconfiguró este visual para crear las diferentes visualizaciones, mostradas más adelante. El ejemplo que este desarrollador creó se encuentra en la carpeta `Testing` de nuestro repositorio, `spherestest.py`. (Ilustración 5).

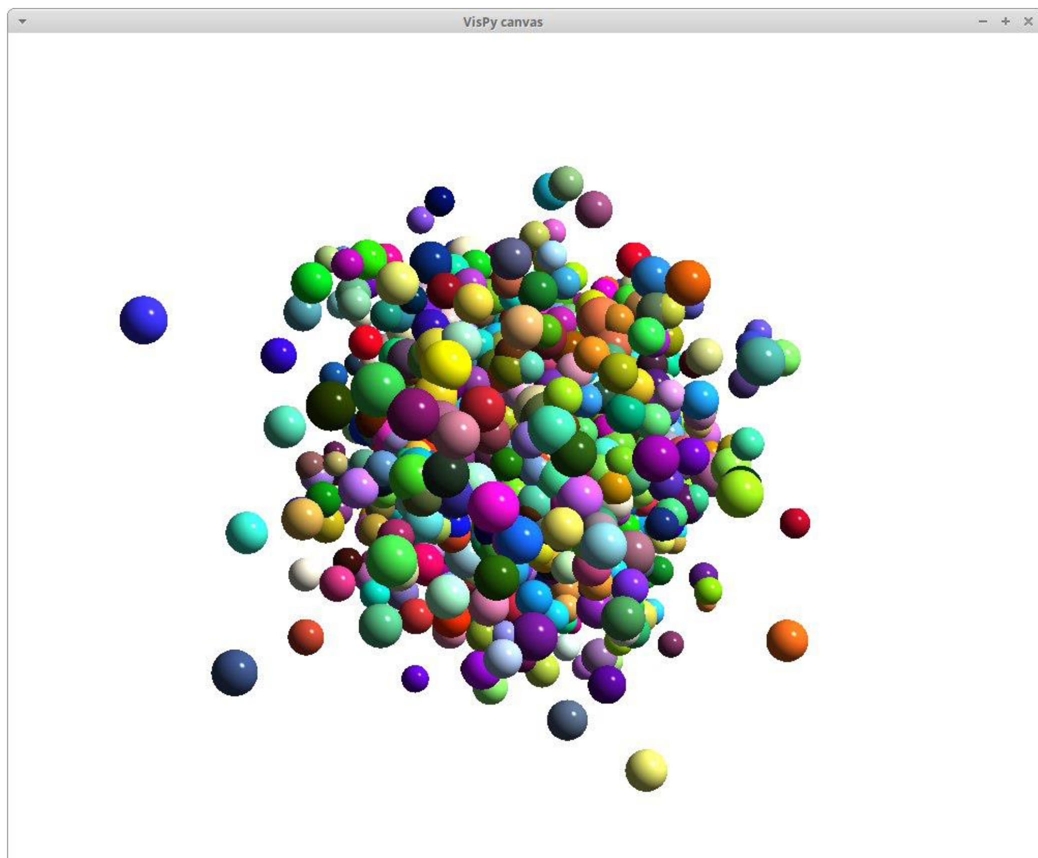

Ilustración 5 Representación de un grupo de esferas aleatorio en `Testing/spherestest.py`, creado por el usuario Luke Campagnola.

Según este usuario, VisPy tiene una forma peculiar de obtener los sistemas de coordenadas en las transformaciones. Estos conocimientos no son tan fáciles de adquirir dado la falta de documentación al respecto, pero gracias a la ayuda personal del desarrollador pudimos entender el proceso:

- `vertex`:
    1. Determinar las posiciones de los átomos en la pantalla, utilizando las transformaciones para su renderizado.
    2. Medir la orientación del sistema de coordenadas `framebuffer` relativo al átomo.
    3. Utilizar el eje x para medir el radio de los píxeles del `framebuffer`.
    4. Utilizar el eje z para medir la posición y radios del `depth buffer`.
- `fragment`:
    1. Calcular las posiciones xyz del `fragment` relativas al radio.
    2. Difundir color para crear las esferas.
    3. Crear la luz especular para dar el efecto de brillo y sombra.
- `class(visuals.Visual):`
    1. Creó funciones que permiten cargar los datos que se aplican a la visual, usando la plantilla creada.
    2. Preparó las transformaciones de `visual` a `framebuffer`, de `framebuffer` a `visual`, y de `framebuffer` a `render`.

La mayor diferencia de este visual con el documento `molecular_viewer.py` es que las esferas de `spherestest.py` tienen profundidad, no son bidimensionales con 3D falso.

Visualizaciones

Antes de crear la escena con nuestro modelo, tenemos que crear los sistemas de datos. Por comodidad, se agruparán en variables de tipo global. Es importante además determinar qué visualización carga qué características, ya que las variables globales serán las mismas para todo.

Es importante crear una función que centre la matriz de posiciones para que se encuentre siempre en el centro de la escena. Un buen ejemplo de esta función es la que se encuentra en `molecular_viewer.py`.

```
def centroid(arr):
    length = arr.shape[0]
    sum_x = np.sum(arr[:, 0])
    sum_y = np.sum(arr[:, 1])
    sum_z = np.sum(arr[:, 2])
    return sum_x/length, sum_y/length, sum_z/length
```

A partir de aquí, podemos crear una función que nos cambie las variables globales según los datos y modos que se apliquen. Por suerte, en VisPy podemos utilizar todos los datos en una sola matriz, no como en matplotlib. Esto facilita mucho la organización.

<u>CPK</u>

Esta es la visualización más sencilla. Tan solo hay que obtener las coordenadas de los átomos, centrarlas, encontrar los colores y los radios. Cada matriz tendrá su tipo de datos.

```
    if mode == 'cpk':
        #list of atoms
        atoms = [atom for atom in structure.get_atoms()]
        natoms = len(atoms)
        #atom coordinates
        coordinates = np.array([atom.coord for atom in atoms])
        center = centroid(coordinates)
        coordinates -= center

        #atom color
        color = [colorrgba(atom.get_id()) for atom in atoms]
        #atom radius
        radius = np.array([vrad(atom.get_id()) for atom in
atoms])
```

<u>Aminoácidos</u>

Esta visualización también es muy sencilla. Se realizan los mismos pasos que con CPK, pero se obtienen los colores a partir del nombre del aminoácido pariente de cada átomo.

```
    elif mode == 'aminoacid':
```

```
        #list of atoms
        atoms = [atom for atom in structure.get_atoms() if 
atom.get_parent().resname != 'HOH']
        natoms = len(atoms)
        #atom coordinates
        coordinates = np.array([atom.coord for atom in atoms])
        center = centroid(coordinates)
        coordinates -= center
        #atom color
        color = [colorrgba(restype(atom.get_parent().resname)) 
for atom in atoms]
        #atom radius
        radius = np.array([vrad(atom.get_id()) for atom in 
atoms])
```

Por cadenas

Aquí se complican un poco las cosas. En esta visualización necesitamos una matriz con las coordenadas de cada átomo, pero también separadas por cadenas. Además, hay que hacer que todos los átomos de una cadena tengan el mismo color que la cadena en sí. Para ello se puede crear una lista de matrices con los colores de cada átomo y cada cadena por separado, pero aprovechando el *loop* de cada cadena para ello. Además, hay que centrar las cadenas de forma individual también.

```
elif mode == 'backbone':
        #list of atoms
        atoms = [atom for atom in structure.get_atoms() if 
atom.get_name() == 'CA' or atom.get_name() == 'N']
        natoms = len(atoms)
        #atom coordinates
        coordinates = np.array([atom.coord for atom in atoms])
        center = centroid(coordinates)
        coordinates -= center
        #atom color
        color = []
        #list of arrays of coordinates and colors for each chain
        chains = []
        chain_colors = []
        chain_coords=[]
        for chain in structure.get_chains():
            chains.append(chain)
            can_coord = np.array([atom.coord for atom in 
chain.get_atoms() if atom.get_name() =='CA' or atom.get_name() 
=='N'])
            can_coord -= center
            chain_coords.append(can_coord)
            chain_length = len(can_coord)
            chain_color = np.append(np.random.rand(1,3),[1.0])
            chain_colors.append(chain_color)
            color.append(np.tile(chain_color,(chain_length,1)))
        if len(chains)>1:
            color = np.concatenate(color)
        #atom radius
        radius = np.array([vrad(atom.get_id()) for atom in 
atoms])
```

DSSP

DSSP es la forma más complicada de visualización en VisPy. La obtención de la lista de átomos, coordenadas, estructura y radio es igual que en el resto.

```
    elif mode == 'dssp':
        #list of atoms
        atoms = [atom for atom in structure.get_atoms() if 
atom.get_name() == 'CA' or atom.get_name() == 'N']
        natoms = len(atoms)
        #atom coordinates
        coordinates = np.array([atom.coord for atom in atoms])
        center = centroid(coordinates)
        coordinates -= center
        #atom color
        struct3 = [dssp[key][2] for key in list(dssp.keys())]
        residues = [residue for residue in 
structure.get_residues() if residue.get_resname() in 
resdict.keys()]

#[...]

        #atom radius
        radius = np.array([vrad(atom.get_id()) for atom in 
atoms])
```

Pero aquí hay que realizar 2 pasos, la obtención de los colores de cada átomo, y la obtención de los colores de cada cadena. Este último es igual que en la representación anterior, pero para crear la matriz de colores DSSP hay que primero enlazar cada residuo con su representación DSSP, elegir los átomos a representar del residuo y utilizar el color DSSP en los átomos elegidos.

```
#[...]

        color = []
        for i in range(len(struct3)):
            dsspcolor = crgbaDSSP(struct3[i])
            n_atoms = len([atom for atom in residues[i] if 
atom.get_name() =='CA' or atom.get_name() == 'N'])
            color.append(np.tile(dsspcolor,(n_atoms,1)))
        if len(struct3)>1:
            color = np.concatenate(color)
        #list of arrays of coordinates and colors for each chain
        chains = []
        chain_colors = []
        chain_coords =[]
        for chain in structure.get_chains():
            chains.append(chain)
            chain_color = np.append(np.random.rand(1,3),[1.0])
            chain_colors.append(chain_color)
            can_coord = np.array([atom.coord for atom in 
chain.get_atoms() if atom.get_name() =='CA' or atom.get_name()
=='N'])
            can_coord -= center
```

```
            chain_coords.append(can_coord)

#[...]
```

Escena y cámara

Una de las grandes ventajas del uso de visuales es la posibilidad de agrupar las diferentes herramientas y agruparlas en una sola escena. Esta escena puede tener además diferentes tipos de cámaras, ya predeterminadas, eliminando la necesidad de programarla mediante cálculos con matrices **cuaternión**.

Lo primero que hay que hacer una vez se tienen las herramientas es la creación del `canvas`, y añadir una cámara.

```
from vispy import app, gloo, visuals, scene

#[...]

        canvas = scene.SceneCanvas(keys='interactive',
app='pyqt4', bgcolor='white', size=(1200,800), show=True)
        view = canvas.central_widget.add_view()
        view.camera = scene.ArcballCamera(fov=70,
distance=(self.radius+40))
```

En esta parte del código se puede ver que el atributo `distance` tiene un cálculo hecho mediante `self.radius+40`. Este radio se calcula con la coordenada más lejana dentro del sistema de coordenadas que se utilice en el momento mediante:

```
        self.radius = max(abs(np.concatenate(coordinates)))
```

Una vez creada la escena, es importante cargar los visuales como nodos.

```
        Spheres =
scene.visuals.create_visual_node(SpheresVisual)
        Lines =
scene.visuals.create_visual_node(visuals.LinePlotVisual)
```

En este caso, `Spheres` es el visual creado por Luke Campagnola, y `Lines` es un visual que viene por defecto en la librería VisPy.

Una vez cargadas las visuales, podemos aplicarlas. `Lines` solo se aplicará si el modo de visualización es por cadenas o DSSP.

```
#[...]

vis_atoms=[Spheres(coordinates,color,radius,parent=view.scene)]
        vis_chains=[]
        if self.mode in ['backbone','dssp']:
            for i in range(len(chains)):
                vis_chains.append(Lines(chain_coords[i], color =
chain_colors[i],parent=view.scene))
```

```
#[...]
```

Es importante determinar a qué escena está anclada cada visual. Para finalizar, podemos cargar el `canvas` para visualizar el resultado.

```
#[...]

                canvas.app.run()
```

2.4 GUI – Tkinter

Tkinter permite la creación de una GUI de una manera sencilla, y en este caso dividimos la ventana en 4 secciones:

- Explorador de directorios para seleccionar nuestro archivo.
- Selector de librería.
- Selector de visualización.
- Botón de ejecución con ventana de error.

A lo largo del código hay una función llamada `grid()` que nos permite colocar nuestros elementos en una cuadrícula. Dentro de cada elemento también encontramos una variable llamada `frame`, que es en realidad a que parte de la GUI se encuentra anclado el elemento.

Selección de archivo

Nuestro selector de archivos consta de dos partes: El botón que nos abre el explorador, y una etiqueta que nos muestra el archivo seleccionado.

Para poder realizar esto, necesitamos crear una función que contenga el explorador, y a la vez que nos genere una variable de tipo *string* (texto) que pueda cambiar según el directorio seleccionado.

De forma externa a la clase, creamos esta función:

```
from Tkinter import *
import tkFileDialog

def browse_button():

    filename = tkFileDialog.askopenfilename(filetypes = (("pdb files","*.pdb"),("all files","*.*")), title='Select your PDB file')
    folder_path.set(filename)

    try:
        endname = "..." + filename[-9:]
    except:
        endname = filename

    pathname.set(endname)
```

Así además creamos una segunda variable que contenga solo el final, permitiéndonos leer tan solo el nombre del archivo, o parte de él.

Dentro de nuestra aplicación, necesitaremos entonces crear el botón que use esta función y las variables de clase global que utilizará esta función:

```
        global folder_path
        global pathname

        folder_path = StringVar()
        pathname = StringVar()
        pathname.set("...")

#[...]

        self.browseb = Button(frame, text="Browse",
command=browse_button).grid(row=1, column=0,sticky=W)
        self.pathlabel = Label(frame, textvariable=pathname,
width=12).grid(row=1, column=1, sticky=W)
```

Así tenemos la primera parte de la interfaz.

Selección de librería

La selección de la librería es muy sencilla. Se utiliza la función `Radiobutton()` y creamos una variable que pueda cambiar para estos botones: de esta forma quedará almacenado qué se ha seleccionado.

```
        self.ltype = StringVar()
        self.ltype.set("matplot")

        self.lib1 = Radiobutton(frame, text="MatPlot",
variable=self.ltype,
value="matplot").grid(row=3,column=0,sticky=W)
        self.lib2 = Radiobutton(frame, text="VisPy",
variable=self.ltype,
value="vispy").grid(row=3,column=1,sticky=W)
```

Con esto hecho, `self.ltype` tendrá almacenada la selección de librería.

Selección de visualización

Para crear un menú desplegable se tiene que utilizar un paquete distinto a *Tkinter* pero que está incluido: `ttk`. Este paquete incluye una función llamada `OptionMenu` que permite utilizar una lista para crear un botón de menú.

```
        self.vtype = StringVar()
        choices = ['CPK','Residue type','Chains','DSSP program']

        self.vtype.set('CPK')

        self.combo = OptionMenu(frame, self.vtype,
*choices).grid(row=5, columnspan=2,sticky=W)
        self.combo
```

De esta forma, tendremos el tipo de visualización almacenada en `self.vtype`.

Ejecución

La última sección de la interfaz es el botón de ejecución, que es tan solo un botón más, pero que tiene una función vinculada que aprovecha todos los datos almacenados anteriormente. Según la librería almacenada en `self.ltype` usaremos `MatViewer` o `VisPyViewer`, usaremos el archivo almacenado en `filename`, y utilizaremos la visualización almacenada en `self.vtype`.

```
import tkFileDialog

#[...]

    def vis_options(self):

        choices_dict = {'CPK':'cpk','Residue
type':'aminoacid','Chains':'backbone','DSSP program':'dssp'}

        if folder_path.get()[-4:] != '.pdb':
            tkMessageBox.showerror("Error", "You have not
selected a PDB file")

        else:
            if self.ltype.get() == "matplot":
                return
MatViewer(folder_path.get(),choices_dict[self.vtype.get()])

            else:
                return
VisPyViewer(folder_path.get(),choices_dict[self.vtype.get()])
```

Dentro de esta función, además, añadiremos una condición: Si el archivo seleccionado (o no seleccionado) no acaba en '.pdb', nos saldrá una ventana de error. Esto se puede usar como alternativa a `try-except`, para asegurar que no se leerá ningún elemento extraño en el *parser*.

2.5 Ejemplo práctico

Este ejemplo utiliza como archivo PDB 1YD9, la estructura cristalina de un dominio no-histona de la **histona** variante MacroH2A1.1.[58] En el repositorio proporcionado en el Anexo 6.1 Repositorio se puede encontrar este archivo dentro de la carpeta `data`.

Matplotlib

Tras importar `MatViewer` desde `Viewer2d`, tan solo hay que ejecutar la función. Las opciones de visualización son `cpk` (Ilustración 6), `aminoacid` (Ilustración 7), `backbone` (Ilustración 8) y `dssp` (Ilustración 9). Solo se puede seleccionar una cada vez:

```
from Viewer2d.MatViewer import MatViewer
```

```
MatViewer(data/1yd9.pdb, [cpk|aminoacid|backbone|dssp])
```

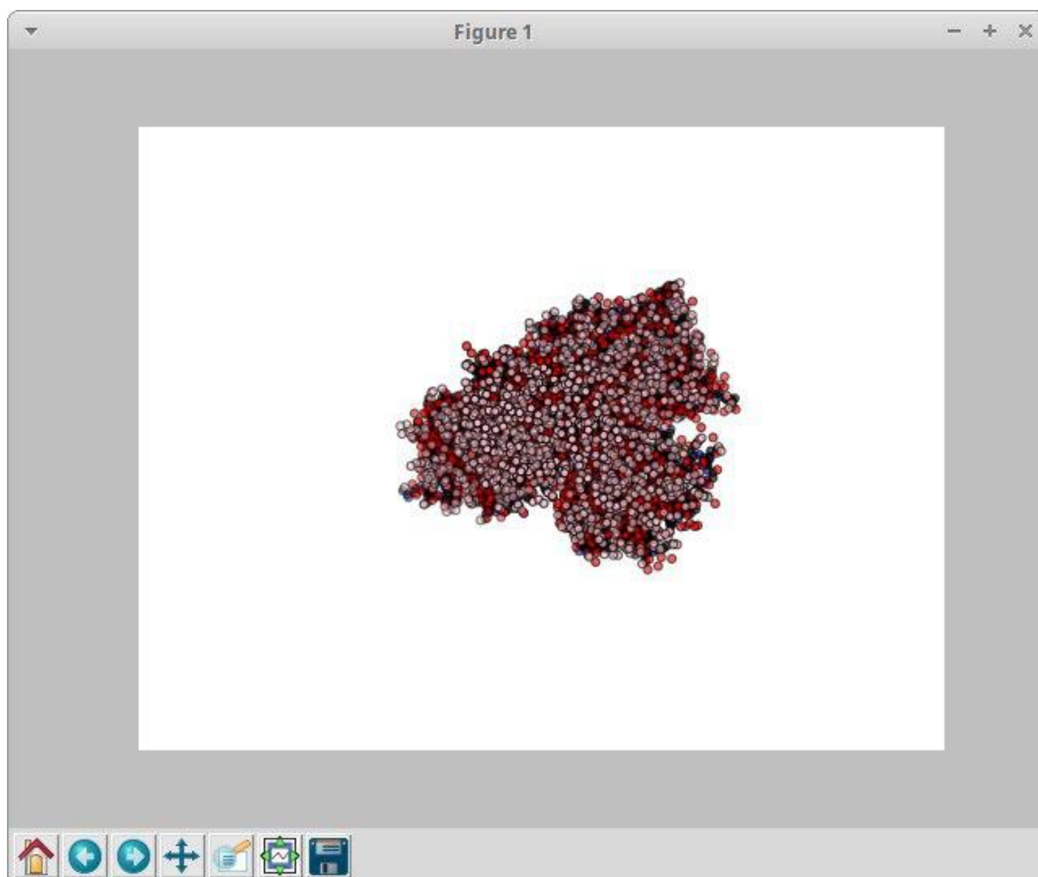

Ilustración 6 Visualización CPK del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`.

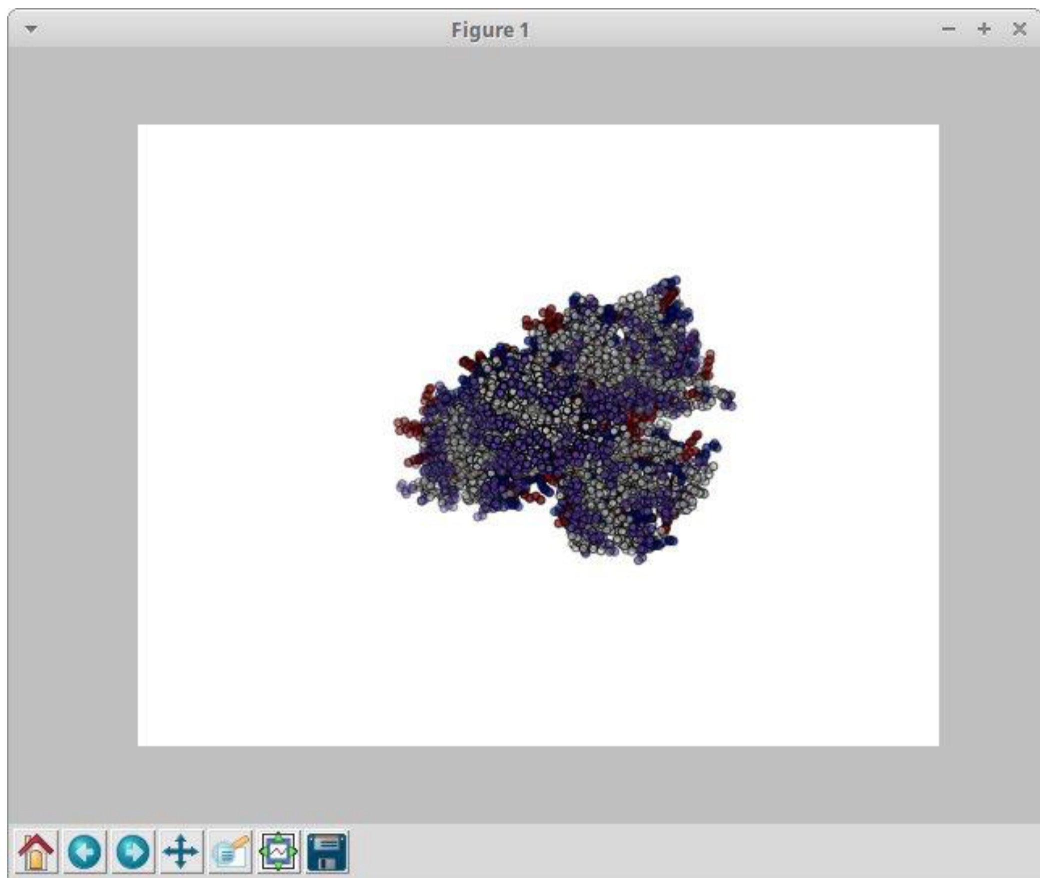

Ilustración 7 Visualización según el tipo de aminoácido del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`.

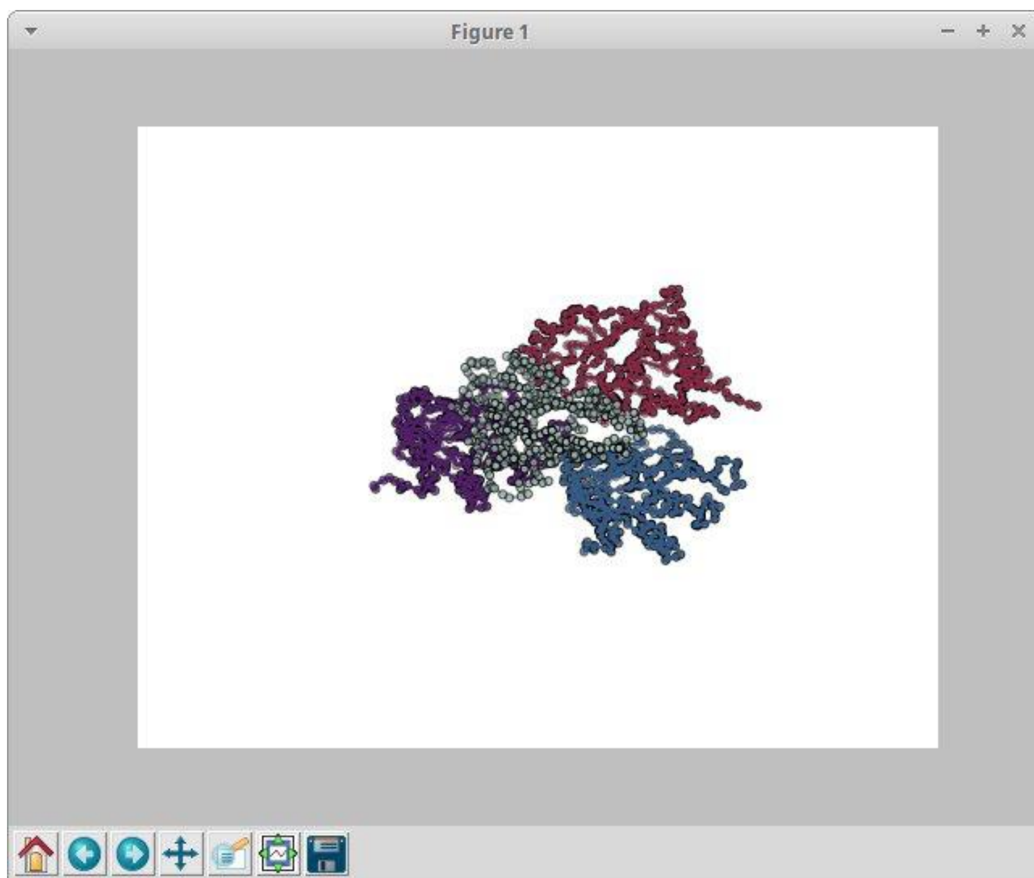

Ilustración 8 Visualización por cadenas del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`.

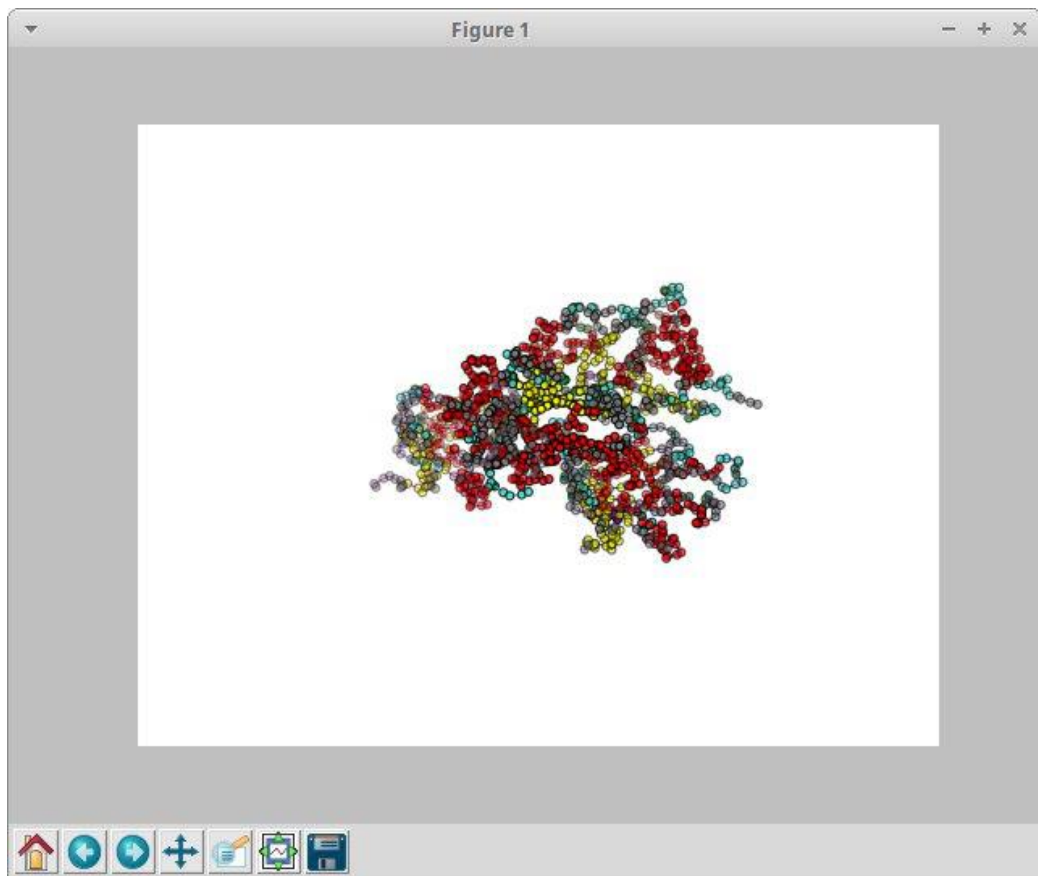

Ilustración 9 Visualización DSSP del archivo `1yd9.pdb` mediante la librería matplotlib aplicada al paquete `Viewer2d.MatViewer`.

Es importante destacar que la cámara se puede controlar con el ratón: el click izquierdo controla la rotación y el click derecho el *zoom*.

VisPy

Tras importar `VisPyViewer` desde `Viewer3d`, tan solo hay que ejecutar la función. Las opciones de visualización son `cpk` (Ilustración 10), `aminoacid` (Ilustración 11), `backbone` (Ilustración 12) y `dssp` (Ilustración 13). Solo se puede seleccionar una cada vez:

```
from Viewer3d.VisPyViewer import VisPyViewer

VisPyViewer(data/1yd9.pdb, [cpk|aminoacid|backbone|dssp])
```

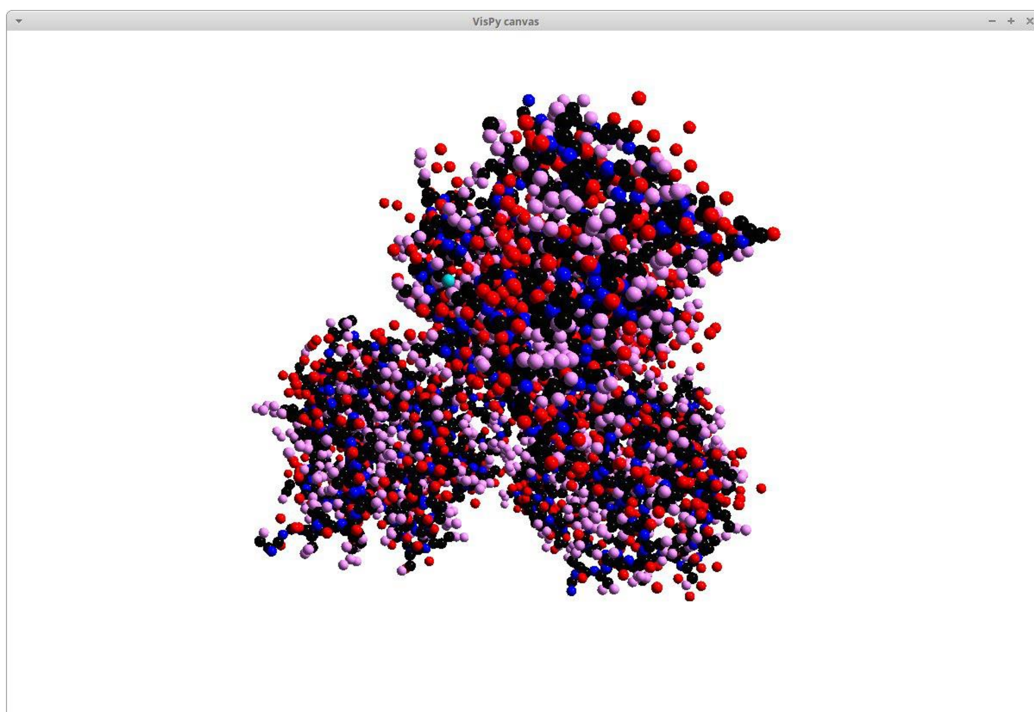

Ilustración 10 Visualización CPK del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`.

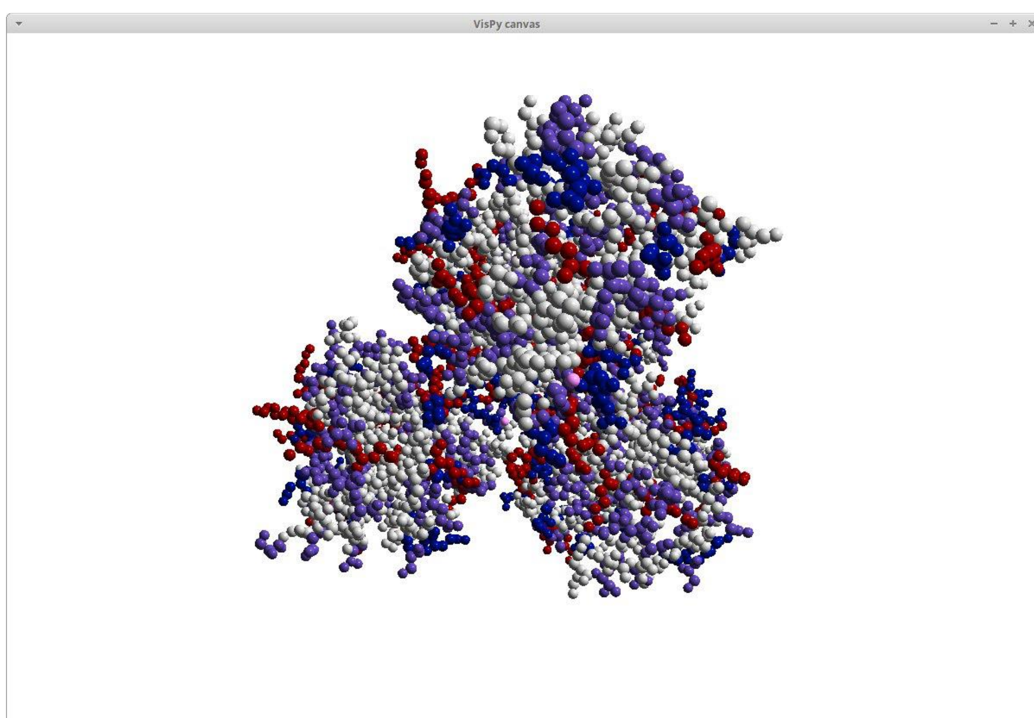

Ilustración 11 Visualización según el tipo de aminoácido del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`.

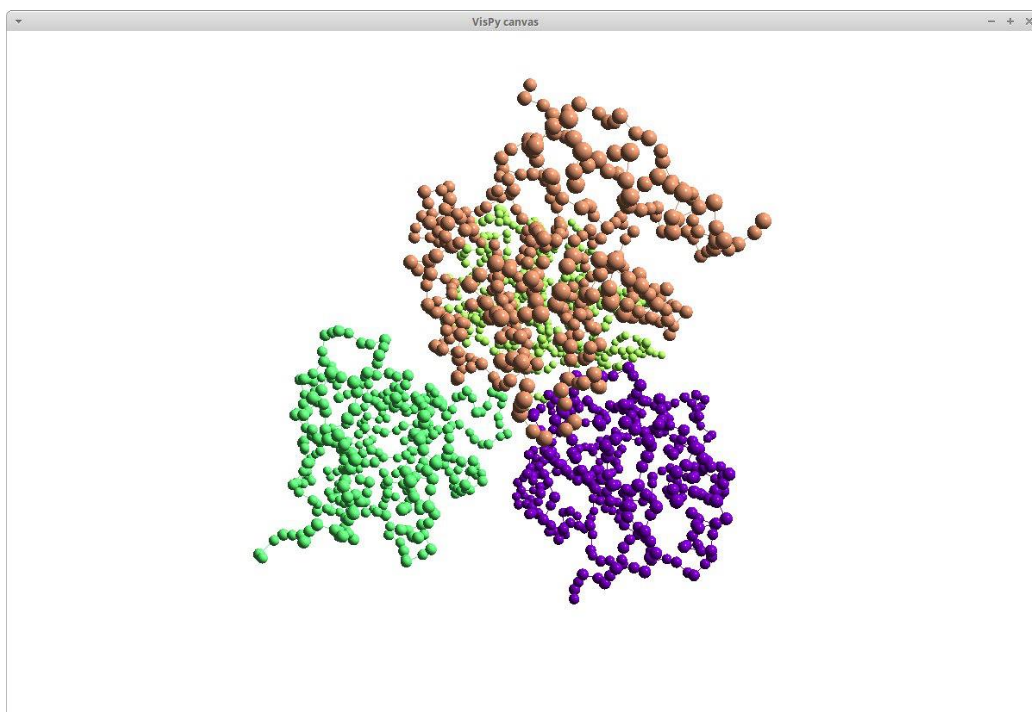

Ilustración 12 Visualización por cadenas del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`.

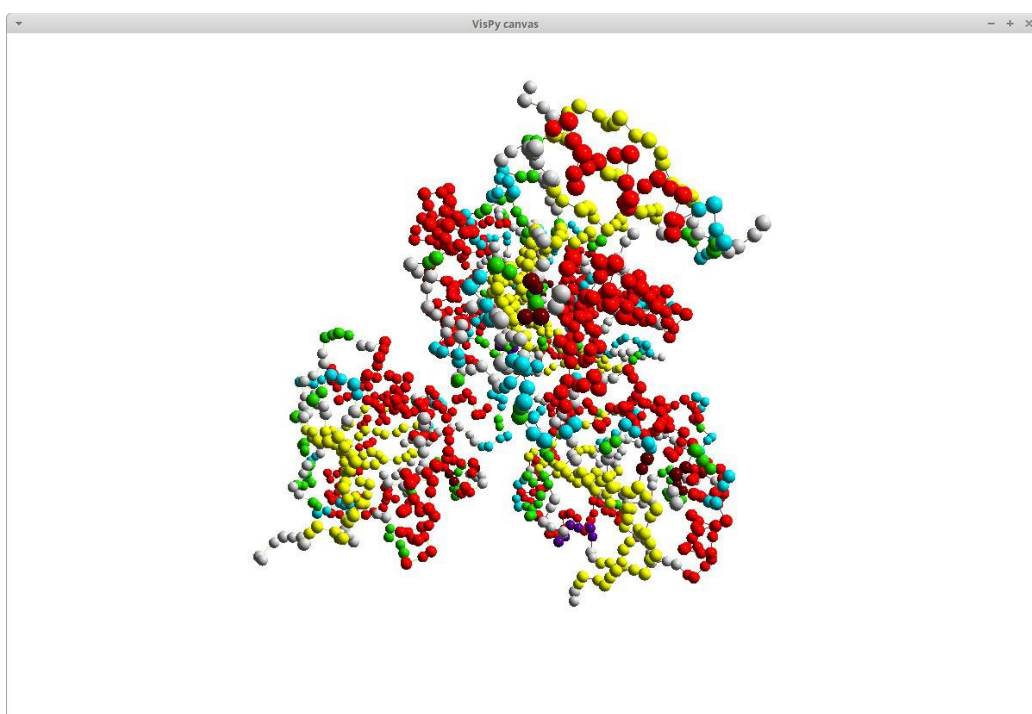

Ilustración 13 Visualización DSSP del archivo `1yd9.pdb` mediante la librería VisPy aplicada al paquete `Viewer3d.VisPyViewer`.

Es importante destacar que la cámara se puede controlar con el ratón: el click izquierdo controla la rotación y el click derecho el *zoom* (la rueda del ratón también sirve). Si se mantiene pulsado el botón Shift usando el ratón, se puede mover la escena en diferentes ejes, y cambiar el campo de visión de la cámara utilizada.

GUI

La ejecución del GUI es muy sencilla. Dentro del directorio del repositorio, hay que ejecutar el comando siguiente:

```
$ python pviewer.py
```

Tras ejecutar esto, nos aparecerá una ventana (Ilustración 14) con las opciones a cambiar. Al pulsar el botón `Browse` nos aparecerá otra ventana para poder elegir el directorio en el que se encuentra el archivo. La búsqueda se puede filtrar según el tipo de archivo. ()

La siguiente opción es la librería que elegir, diferenciando MatPlot y VisPy.

La última opción que elegir es el tipo de visualización, donde podemos hacer click que abrirá un menú para seleccionar una de las cuatro visualizaciones ya mencionadas a lo largo del proyecto.

Una vez seleccionado todo, haremos click en `Run`, pero si el archivo que se ha seleccionado no es de tipo PDB, nos saldrá una ventana de error (Ilustración 16).

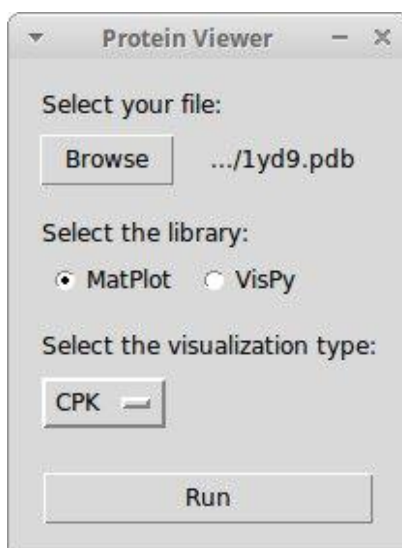

Ilustración 14 GUI presente en `pviewer` creada mediante Tkinter para la combinación de los paquetes `MatViewer` y `VisPyViewer`.

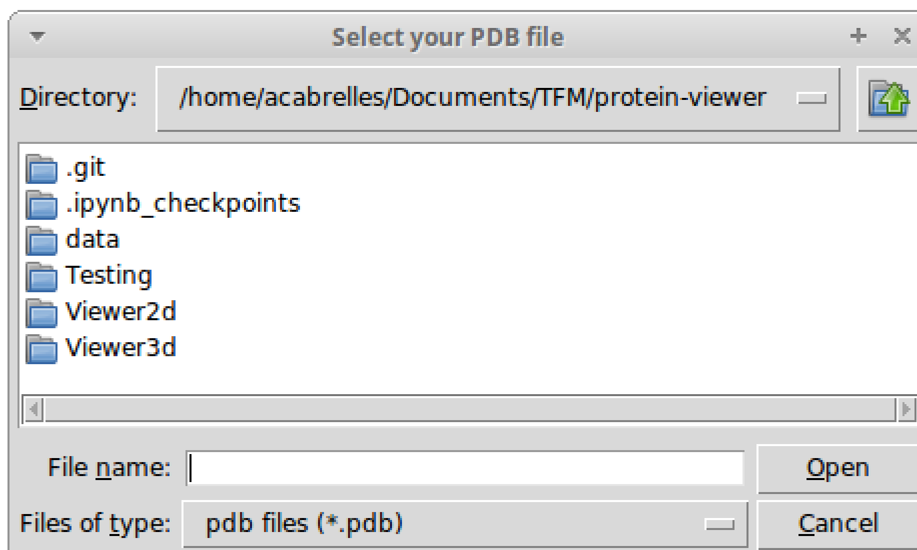

Ilustración 15 Ventana de selección de archivo tras pulsar el botón `Browse` dentro del GUI presente en `pviewer`.

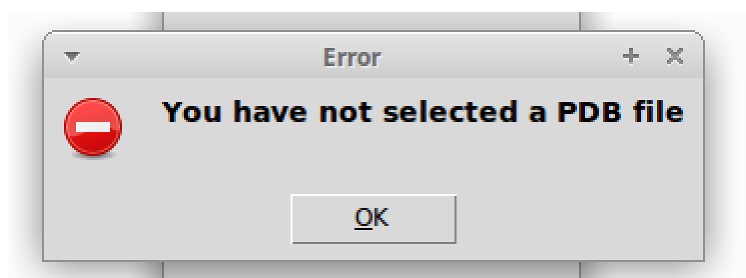

Ilustración 16 Ventana de error que aparece tras pulsar el botón `Run` sin elegir un archivo de tipo PDB dentro del GUI presente en `pviewer`.

3. Conclusiones

A lo largo de este proyecto hemos coordinado el uso de una pequeña variedad de librerías desde el procesado de archivos PDB hasta su visualización en librerías matemáticas y de OpenGL. En este proceso no solo hemos aprendido a usar dichas librerías, sino que también hemos aprendido:

- Instalación de librerías.
- Programación orientada a objetos.
- Creación de interfaces.
- Valoración de pros y contras de cada paso y finalidades de cada librería.
- Organización del código para que sea fácil de entender.
- Compartir y distribuir el código mediante GitHub.
- Búsqueda de código de acceso público.
- Comunicación con otros desarrolladores y corrección mediante críticas y recomendaciones.
- *Debugging*.

Algo importante que comentar sobre el resultado es que es que matplotlib y VisPy tienen dificultades de aprendizaje muy distintas, pero limitaciones completamente diferentes también. Matplotlib permite crear *plots* y gráficos que son muy útiles para artículos e informes, pero su uso no está orientado a la creación de modelos interactivos a gran escala y se nota por su alto gasto a nivel de recursos en el sistema. VisPy, por otro lado, tiene un techo de aprendizaje mucho más elevado, pero permite la creación de modelos no solo personalizables con una gran variedad de funciones y visuales (una infinidad con conocimientos previos de OpenGL al poder crear los *shaders* de forma autónoma), sino que además con una gran interactividad y un bajo consumo de recursos en el sistema.

Es cierto que se han logrado todos los objetivos hasta el final, pero hay partes del código que nos hubiera gustado refinar más con cierto tempo más lento: nos hemos dispersado mucho en diferentes conceptos y nos ha forzado a cometer errores básicos a lo largo del proyecto, tanto a nivel de código como de refinado, complicando ciertos pasos más de la cuenta. Cabe destacar que el enlace polipeptídico formado está basado en Cα-N, y dada la presencia de N en este enlace, es posible que la visualización átomo-átomo genere problemas. Durante el proyecto no se detectó ninguna aberración, así que se decidió mantenerlo como estaba, pero en caso de que se encuentre un error en el futuro, es fácil de arreglar al estar todo sistematizado.

La planificación se siguió desde el principio, pero un momento crítico ha sido la introducción al mundo de OpenGL por VisPy. Sin una previa formación a este lenguaje, la ayuda de los desarrolladores de VisPy (además de su código de acceso público), ha sido crítica en los avances del proyecto. Esto ralentizó mucho el proceso y obligó a priorizar lo esencial del producto final: no se pudo añadir interacciones y funciones

extra que se vieron añadidos en otras iteraciones de visualizaciones parecidas.

Este trabajo tenía otro objetivo, que es el utilizarse como un escalón para desarrollar nuevas funciones y visualizaciones personalizadas a partir de las librerías utilizadas y más del tipo:

- La interfaz es mejorable, es posible explorar esta librería mejor para hacer que la visualización se genere dentro de la misma ventana en lugar de que se genere un pop-up.
- Sobre todo, en VisPy: es una librería muy versátil, capaz de añadir una gran interactividad con el usuario: hay funciones de teclado y ratón que se pueden preestablecer de manera muy sencilla. A medida que evolucione esta librería, habrá limitaciones que dejarán de existir. Por ejemplo, en las visualizaciones por cadenas y DSSP las líneas que unen los átomos son solo de 1 píxel utilizando las funciones básicas de sus visuales, pero esto se arreglará en el futuro permitiendo crear líneas más gruesas (es cierto que hay visuales alternativas que permiten esto ya, pero su uso no es tan sencillo y a veces causa problemas).

Durante la programación en VisPy también descubrimos una alternativa que enfatiza el uso de OpenGL de una forma más orientada a programadores especializados a este lenguaje, llamado Glumpy[59], aunque hay documentación que facilita mucho su uso. Esta librería podría ser una gran alternativa para la creación de modelos, aunque requeriría su exploración para entender si es posible generar modelos tan interactivos como los que se pueden obtener con VisPy.

4. Glosario

Ácido carboxílico: ácido orgánico caracterizado por tener uno o más grupos carboxilos (-COOH).[60]

Back-end: En el procesado de información, el *back-end* incluye la optimización del código y su fase final de generación.[61]

C++: Lenguaje con fines generales para la Programación Orientada a Objetos, creado por Bjarne Stroutstrup.[62]

Carboxamida: Derivado del ácido carboxilo en el que la porción OH del grupo COOH se reemplazó por NH_2.[63]

Catalizador: substancia que permite que una reacción química se lleve a cabo a una velocidad mayor o bajo diferentes condiciones que no serían posible de otra forma.[64]

CPK: convención de colores para distinguir átomos de diferentes elementos químicos en modelos moleculares.[46]

CPU: Unidad Central de Procesamiento que realiza cualquier tipo de cálculo matemático.[65]

Cuaternión: Los cuaterniones son miembros de la división no conmutativa en álgebra, inventado por William Rowan Hamilton, sobre los números reales.[66]

Diferenciación: conjunto de procesos en el un grupo de células, tejidos y estructuras no especializadas alcanzan su forma y funciones adultas.[67]

Enlaces peptídicos: Unión del grupo α-carboxilo de un aminoácido a un grupo α-amino de otro aminoácido.[1]

Fuerzas de Van der Waals: Fuerzas de atracción que actúan en átomos neutrales y moléculas que aparece por la polarización eléctrica inducida en cada una de las partículas por la presencia de otras partículas.[68]

GPU: Unidad de Procesamiento Gráfico que realiza cálculos de propósito específico, repitiendo las mismas operaciones con grandes cantidades de datos.[65]

Hidrolizados: que ha sufrido un proceso de hidrólisis, un proceso químico de descomposición que involucra la separación de una unión y el añadido de un catión de hidrógeno y un anión de hidróxido del agua.[69]

Histona: Proteína simple que contiene muchos grupos básicos y es soluble en agua e insoluble en amoníaco. Se le asocia al ADN del núcleo celular.[70]

Puentes de hidrógeno: Atracción electrostática entre el átomo de hidrógeno de una molécula polar y un átomo pequeño electronegativo.[71]

Puentes disulfuro: Unión covalente entre dos átomos de azufre formado por la oxidación de dos grupos sulfhidrilo (SH), cada uno perteneciente a una molécula de cisteína.[72]

Renderizar: Crear una imagen digital a partir de un modelo.[73]

Rasterización: Proceso en el que los sistemas de monitorizado modernos convierten datos electrónicos o señales a imágenes proyectadas, como video o gráficos estáticos.[74]

Tioéter: Compuesto análogo al éter (dos grupos con carbono unidos por un oxígeno) en el que el oxígeno se ha reemplazado por un azufre.[75]

Tiol: cualquier compuesto análogo al alcohol que reemplaza el oxígeno del grupo hidróxilo (OH) por un azufre.[76]

5. Bibliografía

[1] J. M. Berg, J. L. Tymoczko, L. Stryer, y G. J. Gatto, *Biochemistry*, 7th editio. Kate Ahr Parker, 2012.

[2] «Protein Illustrations and Visualization | ASU - Ask A Biologist». [En línea]. Disponible en: https://askabiologist.asu.edu/venom/protein-art. [Accedido: 04-mar-2018].

[3] L. Liu y T. Wang, «2D representation of protein secondary structure sequences and its applications», *J. Comput. Chem.*, vol. 27, n.º 11, pp. 1119-1124, ago. 2006.

[4] «Molecular Representations». [En línea]. Disponible en: https://www.cgl.ucsf.edu/chimera/1.2065/docs/UsersGuide/representation.html. [Accedido: 04-mar-2018].

[5] «ExPASy: SIB Bioinformatics Resource Portal - Proteomics Tools». [En línea]. Disponible en: https://www.expasy.org/tools/. [Accedido: 04-mar-2018].

[6] «Swiss PDB Viewer - Home». [En línea]. Disponible en: https://spdbv.vital-it.ch/. [Accedido: 04-mar-2018].

[7] «SwissDock - The online docking web server of the Swiss Institute of Bioinformatics - Home». [En línea]. Disponible en: http://www.swissdock.ch/. [Accedido: 04-mar-2018].

[8] V. Zoete, M. A. Cuendet, A. Grosdidier, y O. Michielin, «SwissParam: a fast force field generation tool for small organic molecules.», *J. Comput. Chem.*, vol. 32, n.º 11, pp. 2359-68, ago. 2011.

[9] «Jmol: un visor Java de código abierto para estructuras químicas en tres dimensiones». [En línea]. Disponible en: http://jmol.sourceforge.net/. [Accedido: 04-mar-2018].

[10] «MarvinSpace – High performance 3D molecule visualization tool « ChemAxon – Software for Chemistry and Biology». [En línea]. Disponible en: https://www.chemaxon.com/products/marvin/marvinspace/. [Accedido: 14-oct-2017].

[11] «PyMOL | pymol.org». [En línea]. Disponible en: https://pymol.org/2/. [Accedido: 04-mar-2018].

[12] «Molecular Visualization Freeware». [En línea]. Disponible en: http://www.umass.edu/microbio/rasmol/. [Accedido: 04-mar-2018].

[13] «SRS 3D - Integrating Sequences, Features, and 3D Structures». [En línea]. Disponible en: http://srs3d.org/. [Accedido: 04-mar-2018].

[14] «Cn3D Home Page». [En línea]. Disponible en: https://www.ncbi.nlm.nih.gov/Structure/CN3D/cn3d.shtml. [Accedido: 04-mar-2018].

[15] «UCSF Chimera Home Page». [En línea]. Disponible en: http://www.cgl.ucsf.edu/chimera/. [Accedido: 04-mar-2018].

[16] «Lecture 1: Applications of Python in Bioinformatics: Visualizing Protein - YouTube». [En línea]. Disponible en: https://www.youtube.com/watch?v=GkadFs-igdQ. [Accedido: 04-mar-2018].

[17] «SimTK: ProDy: Protein Dynamics Analysis in Python: Project Home». [En línea]. Disponible en: https://simtk.org/projects/prody. [Accedido: 04-mar-2018].

[18] «MDAnalysis · MDAnalysis». [En línea]. Disponible en:

https://www.mdanalysis.org/. [Accedido: 04-mar-2018].
[19] B. Jimenez, «"Dock It!" Dockit, GitHub». [En línea]. Disponible en: https://github.com/brianjimenez/dockit. [Accedido: 04-mar-2018].
[20] «Home — VisPy». [En línea]. Disponible en: http://vispy.org/. [Accedido: 04-mar-2018].
[21] B. Ho, «"PyBall" pyball, GitHub». .
[22] «Primary and secondary databases ppt by puneet kulyana». [En línea]. Disponible en: https://es.slideshare.net/PuneetKulyana/primary-and-secondary-databases-ppt-by-puneet-kulyana. [Accedido: 28-may-2018].
[23] «WORLDWIDE PDB». [En línea]. Disponible en: http://www.wwpdb.org/. [Accedido: 28-may-2018].
[24] «RCSB Protein Data Bank - RCSB PDB». [En línea]. Disponible en: https://www.rcsb.org/pdb/home/home.do. [Accedido: 04-mar-2018].
[25] «DSSP». [En línea]. Disponible en: https://swift.cmbi.umcn.nl/gv/dssp/. [Accedido: 28-may-2018].
[26] W. G. Touw, C. Baakman, J. Black, T. A. H. Te Beek, E. Krieger, R. P. Joosten, y G. Vriend, «A series of PDB-related databanks for everyday needs», *Nucleic Acids Res.*, vol. 43, 2015.
[27] W. Kabsch y C. Sander, «Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features.», *Biopolymers*, vol. 22, n.º 12, pp. 2577-637, dic. 1983.
[28] J. E. Grayson, *Python and Tkinter programming*. Manning, 2000.
[29] A. Downey, «How to Think Like a Computer Scientist».
[30] «Xubuntu». [En línea]. Disponible en: https://xubuntu.org/. [Accedido: 28-may-2018].
[31] «The leading operating system for PCs, IoT devices, servers and the cloud | Ubuntu». [En línea]. Disponible en: https://www.ubuntu.com/. [Accedido: 28-may-2018].
[32] «Entorno de escritorio Xfce». [En línea]. Disponible en: https://xfce.org/. [Accedido: 28-may-2018].
[33] «A.1 History of the software». [En línea]. Disponible en: https://docs.python.org/2.0/ref/node92.html. [Accedido: 28-may-2018].
[34] «Python For Beginners | Python.org». [En línea]. Disponible en: https://www.python.org/about/gettingstarted/. [Accedido: 28-may-2018].
[35] «Applications for Python | Python.org». [En línea]. Disponible en: https://www.python.org/about/apps/. [Accedido: 28-may-2018].
[36] «General Python FAQ — Python 3.6.5 documentation». [En línea]. Disponible en: https://docs.python.org/3/faq/general.html#how-stable-is-python. [Accedido: 28-may-2018].
[37] «Biopython · Biopython». [En línea]. Disponible en: https://biopython.org/. [Accedido: 28-may-2018].
[38] «Structural Biopython FAQ».
[39] «Module DSSP'». [En línea]. Disponible en: http://biopython.org/DIST/docs/api/Bio.PDB.DSSP%27-module.html. [Accedido: 28-may-2018].
[40] «NumPy — NumPy». [En línea]. Disponible en: http://www.numpy.org/. [Accedido: 28-may-2018].
[41] «Matplotlib: Python plotting — Matplotlib 2.0.2 documentation». [En línea]. Disponible en: https://matplotlib.org/. [Accedido: 04-mar-2018].
[42] «Introduction to Panda3D - Panda3D Manual». [En línea]. Disponible en:

https://www.panda3d.org/manual/index.php/Introduction_to_Panda3D. [Accedido: 28-may-2018].
[43] «Documentation — VisPy». [En línea]. Disponible en: http://vispy.org/documentation.html. [Accedido: 28-may-2018].
[44] «Introduction — PyQt 4.12.1 Reference Guide». [En línea]. Disponible en: http://pyqt.sourceforge.net/Docs/PyQt4/introduction.html. [Accedido: 03-jun-2018].
[45] «TkInter - Python Wiki». [En línea]. Disponible en: https://wiki.python.org/moin/TkInter. [Accedido: 28-may-2018].
[46] «Esquema de colores CPK». [En línea]. Disponible en: https://es.wikipedia.org/wiki/Esquema_de_colores_CPK. [Accedido: 28-may-2018].
[47] «Atomic radii of the elements (data page)». [En línea]. Disponible en: https://en.wikipedia.org/wiki/Atomic_radii_of_the_elements_(data_page). [Accedido: 28-may-2018].
[48] «OpenGL Overview». [En línea]. Disponible en: https://www.opengl.org/about/. [Accedido: 03-jun-2018].
[49] «Cyrille Rossant - The Power of Shaders in Real-Time Graphics Programming». [En línea]. Disponible en: https://cyrille.rossant.net/shaders-opengl/. [Accedido: 03-jun-2018].
[50] «Visuals — VisPy». [En línea]. Disponible en: http://vispy.org/visuals.html. [Accedido: 28-may-2018].
[51] «Scenegraph — VisPy». [En línea]. Disponible en: http://vispy.org/scene.html. [Accedido: 28-may-2018].
[52] «The object oriented OpenGL API (gloo) — VisPy». [En línea]. Disponible en: http://vispy.org/gloo.html. [Accedido: 28-may-2018].
[53] «vispy/examples/tutorial/visuals/T01_basic_visual.py». [En línea]. Disponible en: https://github.com/vispy/vispy/blob/master/examples/tutorial/visuals/T01_basic_visual.py. [Accedido: 28-may-2018].
[54] «https://github.com/liubenyuan/vispy-tutorial/blob/master/02-shaders.md». [En línea]. Disponible en: https://github.com/liubenyuan/vispy-tutorial/blob/master/02-shaders.md. [Accedido: 28-may-2018].
[55] «vispy/examples/demo/gloo/molecular_viewer.py». [En línea]. Disponible en: https://github.com/vispy/vispy/blob/master/examples/demo/gloo/molecular_viewer.py. [Accedido: 28-may-2018].
[56] «vispy/examples/tutorial/visuals/T02_measurements.py». [En línea]. Disponible en: https://github.com/vispy/vispy/blob/master/examples/tutorial/visuals/T02_measurements.py. [Accedido: 28-may-2018].
[57] «GitHub - Luke Campagnola». [En línea]. Disponible en: https://github.com/campagnola. [Accedido: 28-may-2018].
[58] S. Chakravarthy, S. K. Gundimella, C. Caron, P. Y. Perche, J. R. Pehrson, S. Khochbin, y K. Luger, «Structural characterization of the histone variant macroH2A», *Mol.Cell.Biol.*, vol. 25, pp. 7616-7624, 2005.
[59] N. P. Rougier, «Python & OpenGL for Scientific Visualization», 2018. [En línea]. Disponible en: http://www.labri.fr/perso/nrougier/python-opengl/. [Accedido: 28-may-2018].
[60] «carboxilic acid - Wolfarm|Alpha». [En línea]. Disponible en:

http://www.wolframalpha.com/input/?i=carboxilic+acid&rawformassumption=%7B%22C%22,+%22carboxilic+acid%22%7D+-%3E+%7B%22Word%22%7D. [Accedido: 03-jun-2018].
[61] «Operating Systems Notes». [En línea]. Disponible en: http://www.personal.kent.edu/~rmuhamma/Compilers/MyCompiler/phase.htm. [Accedido: 03-jun-2018].
[62] «What is the C++ Programming Language? - Definition from Techopedia». [En línea]. Disponible en: https://www.techopedia.com/definition/26184/c-programming-language. [Accedido: 03-jun-2018].
[63] «Amides and Imides», 1974, pp. 166-173.
[64] «Catalyst | Definition of Catalyst by Merriam-Webster». [En línea]. Disponible en: https://www.merriam-webster.com/dictionary/catalyst. [Accedido: 04-jun-2018].
[65] «En qué se diferencia la GPU de la CPU, explicado en cinco minutos». [En línea]. Disponible en: https://es.gizmodo.com/en-que-se-diferencia-la-gpu-de-la-cpu-explicado-en-cin-1780816080. [Accedido: 04-jun-2018].
[66] «quaternion - Wolfram|Alpha». [En línea]. Disponible en: http://www.wolframalpha.com/input/?i=quaternion&rawformassumption=%7B%22C%22,+%22quaternion%22%7D+-%3E+%7B%22MathWorld%22%7D. [Accedido: 03-jun-2018].
[67] «Differentiation | Definition of Differentiation by Merriam-Webster». [En línea]. Disponible en: https://www.merriam-webster.com/dictionary/differentiation. [Accedido: 04-jun-2018].
[68] «Van Der Waals Forces | Definition of Van Der Waals Forces by Merriam-Webster». [En línea]. Disponible en: https://www.merriam-webster.com/dictionary/van der Waals forces. [Accedido: 04-jun-2018].
[69] «Hydrolysis | Definition of Hydrolysis by Merriam-Webster». [En línea]. Disponible en: https://www.merriam-webster.com/dictionary/hydrolysis. [Accedido: 04-jun-2018].
[70] «histona - Diccionario Académico de la Medicina». [En línea]. Disponible en: http://dic.idiomamedico.net/histona. [Accedido: 03-jun-2018].
[71] «Hydrogen Bond | Definition of Hydrogen Bond by Merriam-Webster». [En línea]. Disponible en: https://www.merriam-webster.com/dictionary/hydrogen bond. [Accedido: 04-jun-2018].
[72] «¿Qué es puentes disulfuro?» [En línea]. Disponible en: https://www.cun.es/diccionario-medico/terminos/puentes-disulfuro. [Accedido: 04-jun-2018].
[73] «¿Qué es un Render? - Arquitectura y Renders». [En línea]. Disponible en: https://www.arqing-mexico.com/renders/qué-es-un-render/. [Accedido: 04-jun-2018].
[74] «What is Rasterization? - Definition from Techopedia». [En línea]. Disponible en: https://www.techopedia.com/definition/13169/rasterization. [Accedido: 04-jun-2018].
[75] «Thioether Medical Definition | Merriam-Webster Medical Dictionary». [En línea]. Disponible en: https://www.merriam-webster.com/medical/thioether. [Accedido: 04-jun-2018].
[76] «Thiol | Definition of Thiol by Merriam-Webster». [En línea]. Disponible en: https://www.merriam-webster.com/dictionary/thiol. [Accedido: 04-jun-2018].

6. Anexos

6.1 Repositorio

El enlace al repositorio donde está el código del programa es el siguiente: https://github.com/acabrelles/protein-viewer

Para poder descargarlo, se puede hacer manualmente desde el enlace, o mediante código *Git*.

Para instalar *Git* en *Ubuntu* se puede hacer desde la consola con el comando siguiente, mediante el paquete `apt` (los pasos `update` y `upgrade` son opcionales pero recomendados):

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install git
```

Una vez se ha instalado *Git*, se debe ejecutar la consola desde el directorio deseado, y a continuación:

```
$ git init
$ git clone https://github.com/acabrelles/protein-viewer
```

Con esto, se tiene una carpeta con el repositorio guardado de una forma local.

6.2 Instalación librerías

Python

Para instalar *Python 2.7* usaremos el paquete `apt` como anteriormente, pero junto a *Python* instalaremos el paquete `pip`, que facilita la instalación de librerías.

```
$ sudo apt-get python2.7 python-pip
```

BioPython

Para instalar *BioPython*, se utilizará el paquete recién instalado `pip`:

```
$ pip install biopython
```

DSSP

Para instalar DSSP, se utilizará el paquete `apt`:

```
$ sudo apt-get dssp
```

NumPy

Para instalar *NumPy*, se utilizará el paquete `pip`:

```
$ pip install numpy
```

Matplotlib

Para instalar *matplotlib*, se utilizará el paquete `pip`:

```
$ pip install matplotlib
```

VisPy

Para instalar *VisPy*, se utilizará el paquete `pip`:

```
$ pip install vispy
```

PyQT4

Para instalar *PyQT4*, se utilizará el paquete `apt`:

```
$ sudo apt-get python-qt4
```

Tkinter

Para instalar *Tkinter*, se utilizará el paquete `apt`:

```
$ sudo apt-get install python-tk
```