Universitat de les Illes Balears

escola politècnica superior

Trabajo Final de Grado

GRADO DE INGENIERÍA INFORMÁTICA

Librería de Python para el trazamiento y la animación de trayectorias de GPS almacenadas en ficheros con formato GPX

JUAN JOSÉ MARTÍN MIRALLES

Tutor
Dr. Isaac Lera Castro

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, junio de 2017

GRADO DE INGENIERÍA INFORMÁTICA

Librería de Python para el trazamiento y la animación de trayectorias de GPS almacenadas en ficheros con formato GPX

JUAN JOSÉ MARTÍN MIRALLES

Tutor
Dr. Isaac Lera Castro

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, junio de 2017

A mi familia y, sobre todo, a Espe por darme todo el cariño y las fuerzas para conseguir todo lo que me propongo en mi vida.

A Edu, Paula y Sam, por escucharme durante 4 meses hablar únicamente de este proyecto y, aún así, seguir a mi lado.

A Carlos y Camila, por una carrera increíble que hubiera sido más difícil sin vosotros.

A mi abuelo por ser, simplemente, el mejor abuelo que ha podido existir en este mundo.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

ÍNDICE DE ALGORITMOS

ACRÓNIMOS

GPS Global Positioning System

GPX GPS eXchange Format

CSV Comma-Separated Values

API Application Programming Interface

GIS Geographic Information System

XML Extensible Markup Language

OGC Open Geospatial Consortium

UTC Coordinated Universal Time

JSON JavaScript Object Notation

FPS Frames Per Second

HTML HyperText Markup Language

RGB Red, Green, Blue

ABSTRACT

Debido al gran crecimiento que ha tenido en los últimos años la recopilación de todo tipo de datos, es más necesario tener herramientas y programas que faciliten su análisis de una forma rápida y eficiente. Tanto a nivel tecnológico como a nivel aplicativo, los avances en recopilación de datos de **Sistemas de Posicionamiento Global (GPS)** se ha hecho de cada vez más grande. Por una parte están los móviles, las pulseras y los relojes inteligentes que han ido mejorando la precisión y la capacidad de captura de los datos de GPS. Por otro lado, aplicaciones como Google Maps han posibilitado que la gestión y almacenamiento de estos datos sea más fácil para los usuarios.

Track Animation es la librería de Python creada en este proyecto para cubrir la necesidad de analizar y visualizar trayectorias de GPS de una forma más amigable y sencilla. Permite importar datos de GPS almacenados en ficheros con formato **GPX** y CSV para, posteriormente, analizarlos en base a la duración de las trayectorias, la elevación, la velocidad o cualquier indicador personalizado por el usuario mediante un degradado de colores. Además, si lo que se desea es una visualización más detallada de las rutas, existe la posibilidad de situarlas sobre un **mapa interactivo** en formato HTML.

En este documento se explica cómo se ha implementado la librería *Track Animation* y cómo funciona, detallando las diferentes posibilidades que existen para crear una animación con todas las trayectorias que se deseen y lo que un usuario debería tener en cuenta para procesar los datos de una manera rápida y eficiente.

CAPÍTULO 1

INTRODUCCIÓN

Uno de los grandes retos que hay hoy en día es el de ser capaces de convertir en conocimiento toda la gran cantidad de datos geolocalizados que se generan con los Sistemas de Posicionamiento Global (GPS). Se almacena información geoposicionada muy diversa, desde el tráfico diario de vehículos [3], movimientos de jugadores de fútbol sobre el campo [4], la preservación de espacios naturales [5] o la detección de especies invasoras [6], el estudio de patrones de movimiento [7], de aglomeraciones en eventos deportivos [8] o los puntos de interés que más visitan las personas [9]. La visualización es una forma muy eficaz de interpretar el gran volumen de datos generados en estos y otros escenarios relacionados.

Las nuevas tecnologías como los móviles y relojes inteligentes, entre otros, han posibilitado que sea realmente fácil registrar información geolocalizada, siendo el ámbito deportivo el que ha tenido mayor popularidad [10, 7]. Cabe decir que anteriormente ya existían dispositivos GPS, como los de las marcas Garmin [11] o Tomtom [12], pero el bajo número de usuarios que los usaban provocaba que el número de trayectorias registradas fuera relativamente escaso. Con el abaratamiento de los GPS debido al auge de los móviles inteligentes, se han tenido que reinventar y aún, hoy en día, siguen sobreviviendo con mapas gratuitos actualizados regularmente y con mejor hardware. Es tal el avance tecnológico, que han surgido pequeños prototipos de GPS para realizar estudios como, por ejemplo, SEAL [13], un aparato capaz de capturar datos de GPS y almacenarlos en formato GPX [14] equipado con diferentes sensores de CO2 o NH3.

Paralelamente, en el ámbito del software, varias páginas webs como Bikely [15], Wikiloc [16] o Google Maps han hecho que los usuarios pudieran guardar, compartir y visualizar fácilmente toda esa información capturada por los GPS. Por ejemplo, GeoLife [17] fue una aplicación web en la que se podían subir rutas para procesarlas y mostrarlas sobre un mapa con información estadística adicional acerca de estas. También, aplicaciones para móviles Android y IPhone como *GPX Viewer* [18] son capaces de leer fuentes de datos de GPS de Google Maps, Mapbox, HERE, Thunderforest y OpenStreetMap, además de ficheros con formatos gpx, kml, kmz y loc, y mostrar diferentes indicadores y gráficos de estos.

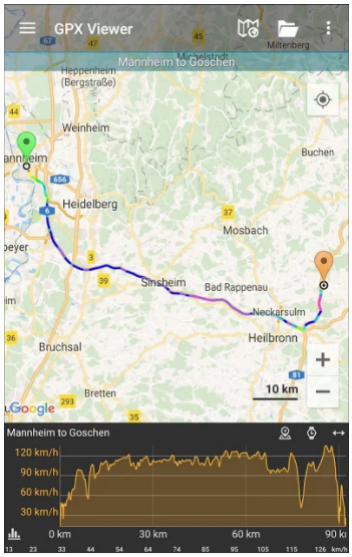


Figura 1.1: Aplicación *GPX Viewer* para el análisis y visualización de datos de GPS

Hoy en día, no es raro encontrar varias aplicaciones y fuentes de datos capaces de guardar, procesar y mostrar toda esta información de GPS usando diferentes métricas como la altitud, la velocidad, el ritmo cardiaco, la potencia o la temperatura del aire, entre otras.

Track Animation es la librería creada en este proyecto y nace con la idea de realizar análisis más personalizados y simples a través de visualizaciones como imágenes o vídeos. Es una librería implementada en Python con la que fácilmente es posible importar datos de GPS provenientes de ficheros con formato GPX [14] o CSV, procesarlos y añadirle métricas personalizadas para crear un vídeo, imagen o mapa interactivo de recorridos fácil de interpretar. Cualquier tipo de usuario, aunque sea con pocos conocimientos de programación, puede crear sus propios análisis de trayectorias, ya que el principal objetivo de la implementación de esta librería es darle un alto nivel de usabilidad y facilitar su uso mediante un diseño exhaustivo de su API para restar a los usuarios finales de implementaciones y análisis complejos para visualizar sus trayectorias.

Los requerimientos esenciales que cumple *Track Animation*, y que están explicados con detalle en sus respectivos capítulos de este documento, son:

1. Importar las trayectorias de un fichero o un grupo de ficheros en formato GPX o CSV.

2. Unir varios conjuntos de trayectorias diferentes con el fin de generar una única visualización con ellas.

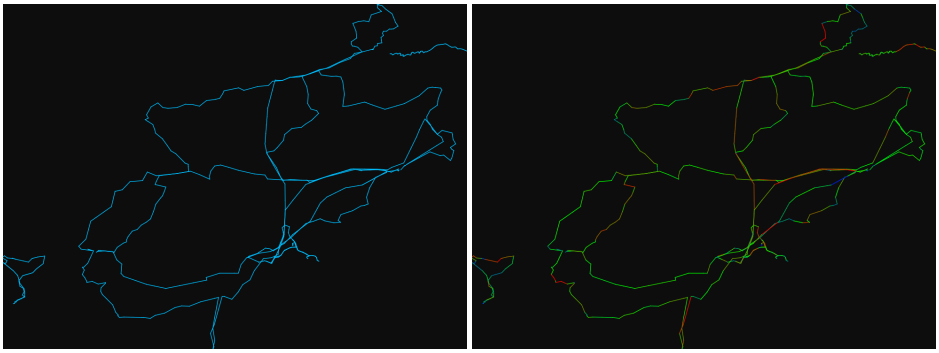
3. Borrar puntos duplicados de una trayectoria a partir de su código, latitud y longitud.

4. Normalizar las trayectorias de manera que todas empiecen y acaben al mismo tiempo especificando la duración del vídeo que se quiere exportar y la cantidad de fotogramas por segundo que ha de tener.

5. Filtrar las trayectorias por lugar, fecha o tiempo.

6. Usar una paleta de colores para asignar a cada uno de los puntos que componen una trayectoria un color a partir de un indicador que desee el usuario, ya sea uno de los que se calculan al importar los ficheros, como la velocidad o la altitud, o cualquiera calculado por él mismo.

7. Exportar una visualización de las trayectorias en formato vídeo (usando *FFMPEG* [19]), en formato imagen o con un mapa interactivo de *OpenStreetMap* en formato HTML.



(a) Resultado básico (b) Resultado con colores según velocidad

(c) Resultado básico con mapa de fondo

Figura 1.2: Ejemplo de diferentes visualizaciones que se pueden crear con *Track Animation*

La librería *Track Animation* está implementada en Python y, al igual que su lenguaje, sigue con la misma filosofía integrando librerías de terceros. Una de estas librerías es Pandas [20], la cual sirve para facilitar la manipulación y el análisis de datos. Consta de dos tipos de estructuras: series y dataframes. Estos últimos son la base de todo el procesamiento de trayectorias de *Track Animation* ya que permiten, con gran facilidad, manipular los datos mediante operaciones numéricas y series temporales [21]. Otra de las librerías usadas es *matplotlib* [22], y sirve generalmente para crear figuras y gráficas muy personalizadas por el usuario final. El uso que se le ha dado en *Track Animation* es

el de visualizar sobre una gráfica cada uno de los puntos de las trayectorias procesadas utilizando las coordenadas geográficas de estos.

Una de las mayores dificultades del proyecto ha sido la implementación de un algoritmo para el cómputo óptimo y eficiente del elevado número de puntos que conforman las trayectorias. Esto es debido a que, en las primeras versiones que se hicieron del algoritmo, para formar un vídeo, se generaba una imagen por cada uno de los puntos para, finalmente, unirlas. Esto conllevaba a una gran cantidad de salidas a memoria secundaria, por lo que su tiempo de ejecución era ser realmente largo e inadecuado. Mediante diferentes técnicas, que se explican a lo largo de este documento, se ha conseguido reducir cuantiosamente el tiempo de ejecución, por ejemplo, de 17 minutos a 1 minuto y medio para un conjunto de datos de 691 puntos.

La precisión con la que los diferentes GPS recogen los datos y los intervalos de captura de cada uno de ellos es otro factor realmente importante para la librería y, por ello, ha conllevado un estudio preliminar. Uno de los ámbitos en los que se han realizado diferentes estudios sobre este tema es en el deportivo y, sobre todo, en los deportes de equipo, como el rugby o el fútbol, en los que hay aceleraciones y desaceleraciones con muy poco tiempo entre ellas y donde un intervalo muy amplio entre una toma y otra de GPS provocaría una gran pérdida de información. Uno de estos estudios es [23], donde comparan 2 dispositivos GPS (SPI-Pro, GPS-5 Hz and MinimaxX, GPS-10 Hz) con un radar como medida estándar, usando diferentes métricas como la distancia, la velocidad y la energía metabólica. Los autores llegaron a la conclusión de que la precisión de un GPS aumenta con una mayor frecuencia de muestreo pero disminuye a medida que se incrementa la velocidad, además de que ambos GPS pueden usarse para calcular la distancia o la media de energía metabólica, pero solo el de 10 Hz es capaz de cuantificar las distancias cubiertas a grandes velocidades.

El proyecto *Track Animation* se ha realizado en 6 fases. En la primera, donde ya se conocía el objetivo principal de la librería, se hizo una búsqueda de la bibliografía y proyectos relacionados para poder conocer lo que se ha llevado a cabo sobre GPS en los últimos años y, más concretamente, en trazado y animación de trayectorias. Posteriormente, en la segunda fase se llevó a cabo un aprendizaje de las principales librerías que componen *Track Animation*: *gpxpy*, *pandas* y *matplotlib*. En la tercera fase se hicieron pequeñas pruebas e implementaciones para saber el alcance que se podía conseguir con cada una de las librerías y los costes computacionales de ciertas funciones. La cuarta fase se basó en la recopilación de los requisitos de la librería *Track Animation* y el comienzo de su implementación. En la quinta fase, momento en el que ya estaba consolidada la implementación de la librería, se añadieron más funcionalidades que permitían una mayor usabilidad y personalización en las visualizaciones finales. Por último, en la sexta fase se escribió la documentación del proyecto y se mejoraron ciertas funciones, tanto a nivel de legibilidad de código como a nivel computacional.

El objetivo del presente documento es explicar cómo se ha implementado la librería *Track Animation* y cuáles han sido los puntos de investigación, tanto a nivel de datos de GPS como a nivel de usabilidad para cualquier usuario, que han llevado a que la librería se implemente de la manera en la que se ha hecho. Además, se analiza y detalla las mejores formas en la que puede ser utilizada para que sea eficiente y fácil de usar. Este documento se compone de diez capítulos de los cuales, **del cuarto al noveno, explican cada uno de los requerimientos que componen la librería** *Track Animation*. Estos requerimientos están introducidos en el apartado 3.1, un vez explicado el conocimiento

necesario para entender el proyecto. Estos capítulos son:

1. **Introducción**: se muestran y describen varios proyectos relaciones con *Track Animation*, tanto a nivel de hardware como a nivel de software, se detallan los objetivos por los que se ha creado este proyecto y da una breve explicación de sus requerimientos y sus puntos fuertes y débiles.

2. **Sistemas de Posicionamiento Global**: debido a que el proyecto se basa en exportar visualizaciones a partir de datos de GPS, es necesario dar una pequeña introducción de qué es un GPS, cuáles son sus características y en qué tipos de ficheros o bases de datos se pueden guardar su información. Además, dado que la principal fuente de importación de datos de *Track Animation* son los ficheros GPX, se explica con detalle qué son y cómo están compuestos.

3. **Arquitectura de la librería *Track Animation***: uno de los objetivos de este proyecto es dar un alto nivel de usabilidad a la librería *Track Animation* para que cualquier usuario esté exento de dificultades técnicas para generar una visualización. Por este motivo, este capítulo va enfocado a explicar en qué paquetes, módulos y clases está compuesta la librería *Track Animation* y cuáles son las librerías de terceros que esta usa para que las explicaciones de algoritmos de los siguientes capítulos puedan comprenderse con más facilidad.

4. **Importación de datos GPS a *Track Animation***: en este cuarto capítulo se explica la primera parte del flujo por el que pasan los datos de GPS en *Track Animation*: la lectura de los ficheros GPX y CSV.

5. **Funcionalidades básicas de la librería *Track Animation***: la exportación de trayectorias a CSV y JSON, la unión de varios conjuntos de datos y el borrado de puntos duplicados son las tres funciones básicas de *Track Animation* que se explican en este capítulo.

6. **Normalización de trayectorias GPS**: la normalización de trayectorias es uno de los requisitos más importantes de la librería *Track Animation* ya que permite que un usuario pueda especificar qué duración desea que tenga un vídeo para que todas las trayectorias que lo componen empiecen y acaben al mismo tiempo. De esta forma se evitan los problemas que el muestreo de coordenadas puede provocar en las visualizaciones. Además, en caso de que sea necesario, se añaden nuevos puntos a las trayectorias con el fin de que en los vídeos se cree un efecto de continuidad y uniformidad al visualizarlas.

7. **Filtrados de trayectorias de GPS en *Track Animation***: con el fin de dar al usuario final una mayor usabilidad y personalización de las visualizaciones, en *Track Animation* es posible filtrar las trayectorias por lugar, fecha y tiempo. Este capítulo está enfocado a explicar con detalle cómo hacer uso de las funciones implementadas para este requisito y mostrar varios ejemplos de ello.

8. **Aplicación de degradado de colores a las trayectorias**: otra de las características de *Track Animation* es dibujar las trayectorias en las visualizaciones con degradados de colores según los valores de un indicador, ya sea uno proveniente de la importación de datos de los ficheros o uno propio creado por el usuario.

9. **Visualización de las trayectorias procesadas en *Track Animation***: el punto final del flujo de datos que pasan por *Track Animation* es la visualización de las trayectorias. Tal y como se explica en este noveno capítulo, estas pueden ser de vídeo, imagen o mapa interactivo, basado en dos algoritmos diseñados para que la generación final de las visualizaciones sea óptima y eficiente como para procesar grandes cantidades de trayectorias.

10. **Conclusión**: en este último capítulo se pueden ver las conclusiones finales del proyecto, los hitos alcanzados y el futuro trabajo para su continuación.

CAPÍTULO 2

SISTEMAS DE POSICIONAMIENTO GLOBAL

Los Sistemas de Posicionamiento Global (GPS) permiten determinar la posición de un objeto en la Tierra desde unos pocos metros hasta 20 o 30 centímetros de precisión si este está cerca de una estación de referencia [24]. La trilateración es uno de los métodos matemáticos utilizados para determinar las posiciones relativas de objetos usando dos o más puntos de referencia (satélites visibles), y la distancia medida entre el objeto y cada punto de referencia. Con tres satélites es posible determinar una posición de dos dimensiones en la Tierra mediante la longitud y la latitud, mientras que con un cuarto satélite se podría conocer la altura a la que está un objeto.

Un *Sistema de Información Geográfica (GIS)* es un conjunto de herramientas que permiten la manipulación, el análisis y la modelización de cualquier tipo de información geográficamente referenciada. Para ello los datos deben estar almacenados en ficheros o bases de datos [25] entre los que se encuentran:

1. Shapefile: es un conjunto de mínimo tres archivos (*shp*, *shx* y *dbf*) que un programa GIS lee como uno único.

2. Bases de datos espaciales que pueden ser *geodatabase personal* o de *archivos de ESRI*, *PostgreSQL + PostGIS*, *Oracle Spatial* o *mySQL*, entre otras.

3. DWG: formato de CAD utilizado principalmente por el programa AutoCAD.

4. GML: sirve para representar información de elementos espaciales a partir de un formato XML.

5. KML: también basado en XML y desarrollado para Google Earth, en 2008 se convirtió en el estándar de la *OGC*. Sirve para representar datos geográficos en tres dimensiones.

6. GPX: GPS Exchange Format puede ser usado sin la necesidad de pagar ninguna licencia para definir *puntos de interés*, *rutas* o *trayectorias* sobre un esquema XML

representando también datos geográficos en tres dimensiones. Su gran ventaja es que pueden ser intercambiados entre diferentes dispositivos GPS.

El objetivo de la librería *Track Animation* es la de crear vídeos, imágenes o mapas de recorridos guardados en un formato de datos de GPS. El formato GPX es el que se ha elegido para la importación principal de estos a *Track Animation* por su facilidad de uso y su flexibilidad para ser intercambiados entre diferentes dispositivos de GPS.

Esta capítulo tiene la finalidad de ser una breve introducción a los Sistemas de Posicionamiento Global (GPS) y al formato de fichero GPX para almacenar sus datos geoposicionados. Está compuesto de tres apartados. El primero de ellos da una definición de *Waypoints*, *Tracks* y *Routes*, elementos de datos GPS que se pueden encontrar dentro de un fichero GPX. El segundo explica cómo están compuestos los ficheros GPX y cómo está estructurada la información que se almacena en ellos. La tercera y última sección da un breve repaso a varias librerías y aplicaciones que son compatibles con los ficheros GPX a día de hoy.

2.1 Puntos de interés, trayectorias y rutas de los ficheros con formato GPX

El formato GPX (*GPS eXchange Format*) ha sido el elegido para la importación de datos de GPS a la librería *Track Animation* debido a que es software libre, no se paga ningún tipo de licencia, está basado en un formato XML, con lo que es muy simple de parsear, y puede ser intercambiado fácilmente entre distintos dispositivos GPS. La primera versión 1.0 apareció en 2002 y la 1.1, que es la que se usa desde entonces como estándar, surgió en 2004 [26].

Los ficheros GPX se pueden componer de hasta tres tipos de objetos: los puntos de interés (Waypoints), las trayectorias (Tracks) y las rutas (Routes).

- ***WaypointType* (wpt)**: es una posición dentro de una colección de puntos que no tienen una relación secuencial y que tiene como finalidad indicar un punto de interés en una trayectoria.

- ***TrackType* (trk)**: es una lista ordenada de puntos que describen un camino formado por, al menos, un segmento. Este segmento tiene una lista de puntos (*TrackPoints*) que están conectados. En caso de pérdida de conexión del GPS, por ejemplo, se crea un nuevo segmento dentro de la trayectoria para continuar con el camino.

- ***RouteType* (rte)**: es una lista ordenada de puntos de una ruta. La diferencia con las trayectorias es que no almacenan las localizaciones de un recorrido que se ha hecho, sino que las rutas indican una sugerencia de camino que se ha especificado de antemano hasta un destino. No tienen fechas ni tiempos pero sí estimaciones de duración o de altitud.

Para *Track Animation* solo se han tenido en cuenta las trayectorias (**trk**), que incluyen los segmentos y la lista de puntos que contiene cada uno de ellos. No obstante, para próximas versiones, no se descarta que se añadan también los otros dos tipos ya que

la librería *gpxpy* es capaz de leerlos y parsearlos igual que las trayectorias. Con esto se podría analizar cuáles son los puntos de interés más visitados o realizar comparativas entre las rutas preestablecidas y los recorridos originales.

2.2 Composición de los ficheros GPX

El formato GPX es un esquema XML compuesto por una serie de tags comunes para almacenar la mayor cantidad de información posible sobre una trayectoria. El tag principal de un fichero GPX es **gpx**, que contiene información como el nombre, la descripción, el autor, etc. También ha de incluir obligatoriamente la versión de GPX (**version**) con la que se creó el fichero y el creador de este (**creator**).

El tag **gpx** es el que contendría los tres tipos de datos comentados en la sección 2.1, cada cual con sus propios atributos. Una trayectoria contiene una lista de segmentos (**trkseg**), y estos a su vez una lista de puntos (**trkpt**). Una ruta está compuesta por una lista de puntos de ruta y un punto de interés es, únicamente, un punto en el mapa. Los posibles elementos de posición que puede tener cada tipo de punto son los especificados en la tabla 2.1.

Campo	Unidades	Descripción
lat	Grados decimales (WGS84)	Latitud
lon	Grados decimales (WGS84)	Longitud
ele	Metros	Elevación en metros
time	Timestamp	Fecha y hora de creación en formato UTC
magvar	Grados	Variación magnética
geoidheight	Metros	Altura, en metros, del geoide (nivel del mar) sobre la elipse de la tierra.

Cuadro 2.1: Campos de posición de un punto (*TrackPoints*)

La latitud y la longitud son los únicos campos obligatorios de un punto, mientras que los demás son opcionales. Además, puede contener otros elementos que son descriptivos del punto como, por ejemplo, un nombre, una descripción o una url, o información de precisión como la dilución de precisión para especificar el efecto adicional en la geometría de la navegación por satélite en medidas de precisión.

Las fechas y las horas no son de hora local, sino que usan el sistema *Coordinated Universal Time* (UTC) con el formato especificado por la ISO 8601, es decir, sigue el criterio de especificar primero los períodos de tiempo más largos y posteriormente los más cortos. Por ejemplo, para especificar una fecha escribiremos en primer lugar el año, luego el mes y, por último, el día [14].

Un ejemplo de fichero con formato GPX sería el siguiente:

```
<gpx xmlns="http://www.topografix.com/GPX/1/1" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" creator="Wikiloc - http://www.wikiloc.com" version
    ="1.1" xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.
    topografix.com/GPX/1/1/gpx.xsd">
    <trk>
        <name>167 MALLORCA</name>
        <cmt>Ruta corta de la 312 Mallorca</cmt>
        <desc>Ruta corta de la 312 Mallorca</desc>
        <trkseg>
            <trkpt lat="39.807557" lon="3.116873">
                <ele>-0.8</ele>
                <time>2014-04-26T05:11:23Z</time>
            </trkpt>
            <trkpt lat="39.807906" lon="3.116760">
                <ele>0.1</ele>
                <time>2014-04-26T05:11:38Z</time>
            </trkpt>
            <trkpt lat="39.808399" lon="3.116590">
                <ele>-0.3</ele>
                <time>2014-04-26T05:11:51Z</time>
            </trkpt>
            .
            .
            .
        </trkseg>
    </trk>
</gpx>
```

Algoritmo 2.1: Ejemplo de fichero con formato GPX

2.3 Librerías y aplicaciones compatibles con el formato GPX

Dado que GPX se ha convertido en el estándar de intercambio de datos GPS, muchísimas aplicaciones y librerías lo usan para leer, compartir, analizar y visualizar estos datos. Ya se han mencionado en la introducción de este documento algunas aplicaciones o webs compatibles con el formato GPX como, por ejemplo, *Wikilog* [16] o *GPX Viewer* [18]. No obstante, a continuación se explican algunos ejemplos más con los que se pueden ver los diferentes ámbitos en los que se usa este formato:

- **Google Earth**, que aunque desarrolló su propio formato (KML), hizo compatible el uso de GPX.
- **MapSource** [11] es una aplicación de escritorio para transferir ficheros GPX a dispositivos Garmin.

- **Nasa World Wind** [27] es un proyecto de software libre para ver imágenes por satélite del mundo y al que se le pueden añadir distintos formatos de archivos de GPS.
- **Quantum GIS** [28] es un programa para crear, editar, visualizar y analizar información geoespacial.
- **iGo Navigation** [29] es una aplicación de GPS para dispositivos móviles al que se le pueden importar y del que se pueden exportar rutas y trayectorias de GPS.
- **Track Road** [30] es una web con la que se pueden planificar rutas y optimizarlas con el fin de ahorrar combustible, tiempo o dinero.
- **gpxpy** [31] es una librería para Python que lee y parsea los datos de los ficheros GPX y con la que fácilmente se pueden sacar indicadores básicos como, por ejemplo, la distancia o la velocidad entre dos puntos. Es la librería usada en *Track Animation* para importar los datos de GPS.

Además, de librerías o aplicaciones, se pueden encontrar variadas fuentes de datos con información gratuita y libre de uso como *GPX Aviation* [32], *BBBike* [33], *GPS Tracks* [34] o *Trace GPS* [35].

CAPÍTULO 3

ARQUITECTURA DE LA LIBRERÍA *Track Animation*

La librería *Track Animation* se ha creado con el fin de poder realizar un análisis visual de trayectorias de GPS a partir de vídeos, imágenes y mapas interactivos de una manera simple y cómoda para cualquier usuario, ya sea con conocimientos de programación o sin ellos. Dar un alto nivel de usabilidad a la librería es uno de los grandes objetivos que se han tenido en cuenta a la hora de su implementación. Gracias a esto, cualquier usuario está exento de dificultades técnicas para crear un vídeo con sus trayectorias y, además, tiene un gran abanico de posibilidades para personalizarlas. No obstante, sí que es necesario un cierto conocimiento de la API creada en *Track Animation* y la estructuración que esta tiene. Por esta razón, se ha decidido crear este capítulo con el fin de explicar cómo es la arquitectura de toda la librería y qué módulos, clases y funciones se han implementado.

Una librería de Python es simplemente un término que se refiere a una colección de código fuente que ha sido implementado con el objetivo de ser usado por aplicaciones, scripts, otras librerías, etc. para ofrecer una funcionalidad específica. Las librerías pueden contener paquetes de Python, que no son más que directorios que, obligatoriamente, han de contener un fichero *__init__.py*, el cual puede estar vacío o no. Los paquetes sirven para organizar los módulos de Python, que son los ficheros *.py* que contienen el código fuente. No obstante, no es estrictamente necesario que un módulo esté dentro de un paquete.

Track Animation está compuesto por un único paquete de Python llamado *trackanimation*. Este se compone de 3 módulos donde estará todo el código de la librería formado por 3 clases. La estructura es la indicada en la figura 3.1.

Los módulos *tracking* y *animation* se componen por las siguientes clases:

1. **ReadTrack**, dentro del módulo *tracking.py*, sirve para leer y parsear un fichero o un conjunto de ficheros en formato GPX o CSV para ponerlos en un dataframe de la librería *pandas*.

```
└── trackanimation
    ├── __init__.py
    ├── animation.py
    ├── tracking.py
    └── utils.py
```

Figura 3.1: Estructura de los módulos que componen *Track Animation*

2. **DFTrack**, dentro del módulo *tracking.py*, se encarga de tratar y procesar los datos importados al dataframe para normalizarlos, filtrarlos o unirlos con más datos de otra fuente.

3. **AnimationTrack**, dentro del módulo *animation.py*, se usa para generar los vídeos, las imágenes o los mapas interactivos, con la ayuda de la librería *matplotlib*, una vez que los datos han sido leídos y procesados.

El módulo *utils.py* tiene varias funciones concretas que son usadas por la librería y una clase muy simple, llamada *TrackException*, para manejar las excepciones de la librería.

Con esta estructura se puede intuir que la librería está compuesta por una clase para la entrada de datos, otra para el procesamiento de estos y otra para generar una salida, respectivamente.

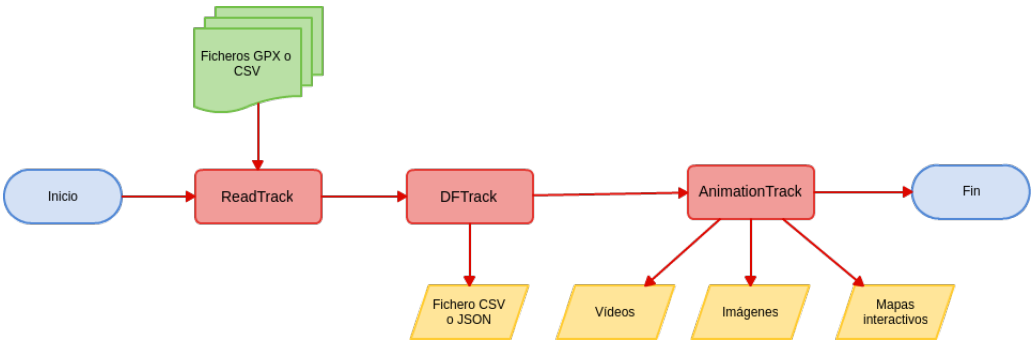


Figura 3.2: Diagrama del flujo de datos en *Track Animation*

Este capítulo va enfocado a explicar de qué paquetes, módulos y clases está compuesta la librería *Track Animation* y cuáles son las librerías de terceros que esta usa para que las explicaciones de los algoritmos de los siguientes capítulos puedan comprenderse con más facilidad. En la primera sección se vuelven a mencionar cuáles son los requerimientos de la librería *Track Animation* para que, en el segundo apartado, se comprenda fácilmente cómo se han usado las librerías de terceros. La tercera, cuarta y quinta sección están destinadas a explicar cada uno de los módulos que componen la librería *Track Animation* con sus respectivas clases implementadas.

3.1 Requerimientos básicos de *Track Animation*

Para entender por qué se ha estructurado de esta manera la librería, es necesario comprender qué se puede hacer con ella y cuáles son sus requerimientos. El objetivo final es que se puedan exportar a vídeos, imágenes o mapas interactivos las trayectorias anteriormente importadas de ficheros con formato GPX. Desde que se importan las trayectorias, hasta que se exportan las visualizaciones, hay un procesamiento de los datos. Este puede consistir en filtrar los puntos por fecha, tiempo o lugar, normalizarlos o ponerles un color según un indicador, ya sea la velocidad, la altitud o cualquiera creado por el usuario.

En resumen, los requerimientos básicos de la librería *Track Animation* son los siguientes:

1. Importar las trayectorias de un fichero o un grupo de ficheros en formato GPX o CSV usando la librería *gpxpy* [31].

2. Unir varios conjuntos de trayectorias diferentes con el fin de generar una única visualización con ellas.

3. Borrar puntos duplicados de una trayectoria a partir de su código, latitud y longitud.

4. Normalizar las trayectorias de manera que todas empiecen y acaben al mismo tiempo especificando la duración del vídeo que se quiere exportar y la cantidad de fotogramas por segundo que ha de tener.

5. Filtrar las trayectorias por lugar, fecha o tiempo mediante la librería *pandas* [20] y *geopy* [36].

6. Dar degradados de colores a cada uno de los puntos que componen una trayectoria a partir del indicador que desee un usuario, ya sea uno de los que se calculan al importar los ficheros, como la velocidad o la altitud, o cualquiera calculado por él mismo.

7. Exportar una visualización de las trayectorias en formato vídeo (usando *FFMPEG* [19]), en formato imagen o con un mapa interactivo de *OpenStreetMap* en formato HTML. Esto se ha hecho utilizando la librería *matplotlib* [22] en la que los puntos se sitúan sobre una gráfica mediante sus coordenadas, y *smopy* [37] para poner como imagen de fondo en la gráfica un mapa. La librería *mplleaflet* [38] se ha usado para pasar las gráficas de *matplotlib* a un mapa interactivo.

Además, se ha utilizado la librería *tqdm* [39] para ver el progreso de los algoritmos implementados en la librería cuando se ejecutan por línea de comandos.

3.2 Librerías de terceros utilizadas

Para crear *Track Animation*, se han utilizado varias librerías de terceros de código fuente abierto con el fin de reutilizar código que ya está creado aprovechando la compartición de conocimientos que se hacen gracias al *open source*. Ya se han ido dando pinceladas

en anteriores capítulos del uso que se ha hecho de todas ellas, pero en esta sección se explican con más detalle.

***gpxpy*: lectura y parseado de ficheros GPX [31]**

Para la lectura y el parseado de los ficheros en formato GPX se ha utilizado la librería *gpxpy*, capaz de leer dichos ficheros y parsearlos creando diferentes listas de objetos de puntos de interés, rutas, trayectorias y los puntos que las componen. No obstante, para *Track Animation* solo se han tenido en cuenta las trayectorias.

***pandas*: procesamiento de trayectorias en dataframes [20]**

pandas es una librería para facilitar la manipulación y el análisis de los datos. Consta de dos estructuras de datos: *series* y *dataframes*. Está implementada sobre la librería *NumPy*. Algunas de las muchas características que tiene es que permite agrupar datos, filtrarlos, convertirlos, unirlos o usar series temporales.

pandas es, junto a *matplotlib*, la librería principal de *Track Animation*. Cualquier trayectoria de un fichero de GPX o CSV leído acaba en un dataframe con el que se manipulan sus datos de la manera en la que quiera el usuario para, finalmente, exportar los resultados, tanto en formato vídeo, imagen, etc., como en formato CSV o JSON. El hecho de haber usado esta librería en *Track Animation* es por la versatilidad en el manejo de datos que permite, siendo muy útil para filtrar fácilmente cualquier punto sobre cualquier campo, incluso haciendo uso de las series temporales sobre las fechas o los tiempos. Es por esta razón que, gracias al manejo de las series temporales que tiene implementado, se han podido crear dos algoritmos para filtrar los puntos por fechas o por tiempos. Esto se explica con más detalle en el apartado 7.2.

***matplotlib*: visualización de trayectorias [22]**

matplotlib es una librería que sirve para crear figuras y gráficas personalizadas como histogramas, diagramas de barras, diagramas de puntos, etc. Tal y como se ha dicho, *matplotlib* es la segunda librería más importante de *Track Animation*, ya que permite visualizar cada uno de los puntos ya procesados en el dataframe de *pandas* en una gráfica. A partir de las latitudes y longitudes, une los puntos dando la posibilidad de personalizar la gráfica con distintos tipos de líneas, varios colores, con imágenes de fondo o con diferentes tamaños.

***smopy*: importación de mapas a *matplotlib* [37]**

Para facilitar la interpretación de los vídeos e imágenes exportados, se da la posibilidad de añadir un mapa estático de los lugares por donde pasan las trayectorias. Para ello se ha utilizado la librería *smopy* que permite, a partir de unos límites geográficos, descargar una imagen de un mapa de *OpenStreetMap* y usarla junto a *matplotlib* haciendo coincidir los puntos exactamente por donde pasan.

***mplleaflet*: conversión de *matplotlib* a mapas interactivos [38]**

mplleaflet es una librería que da la posibilidad de exportar figuras de *matplotlib* a HTML poniendo las trayectorias sobre un mapa interactivo de *OpenStreetMap*.

***geopy*: filtrado de trayectorias por lugares [36]**

En *Track Animation* se ha implementado la posibilidad de filtrar las trayectorias por un lugar en concreto con solo indicar su nombre. Para ello se ha utilizado *geopy*.

geopy es una librería que está implementada por diferentes web services de geolocalización. Es muy útil para localizar las coordenadas de direcciones, ciudades o países usando geocoders de terceros y fuentes de datos. Algunos de los geocoders que utiliza, implementados en diferentes clases, son OpenStreetMap Nominatim, ESRI ArcGIS, Google Geocoding API (V3), Baidu Maps, Bing Maps API, OpenMapQuest o geocoder.us. Esta librería también puede calcular la distancia entre dos puntos mediante la fórmula de Vincenty o la fórmula Ortodrómica con el paquete *geopy.distance*.

***PIL*: conversión de *matplotlib* a imágenes [40]**

PIL (*Python Image Library*) es una librería que sirve para interaccionar con imágenes, es decir, abrir, crear, rotar o convertirlas a otros formatos. En *Track Animation* se ha usado el módulo *Image* de esta librería para convertir una figura de *matplotlib* en una imagen en formato PNG para, posteriormente, crear un vídeo.

***tqdm*: barras de progreso sobre consola de comandos [39]**

tqdm es una librería que crea barras de progreso sobre la consola de comandos en los bucles de una aplicación. Intenta predecir el tiempo que tardará en finalizar el bucle y este será más o menos aproximado siempre y cuando todas las iteraciones duren lo mismo y tengan un tiempo constante. Se ha añadido a *Track Animation* para darle información al usuario sobre el progreso que lleva el procesamiento, una aproximación de lo que queda para acabar y, sobre todo, para darle más interacción.

3.3 Módulo *tracking.py*

El módulo *tracking.py* sirve para leer y procesar datos de trayectorias. Está compuesto por dos clases: *ReadTrack*, que permite leer ficheros de GPX y CSV, y *DFTrack*, que permite normalizar, filtrar, concatenar u ordenar trayectorias.

3.3.1 Clase *ReadTrack*

Esta clase es la encarga de leer de un archivo CSV o de un fichero o directorio con varios ficheros en formato GPX. Utiliza la librería *gpxpy*, la cual lee y parsea los archivos GPX, y la librería *pandas* para leer los CSV.

Está compuesta por 3 funciones:

1. **readGPXFile**: sirve para leer un simple fichero GPX y almacenar sus puntos en una lista.

2. **readGPX**: llama a la función anterior tantas veces como ficheros GPX tenga el directorio pasado por parámetro, o una sola vez en caso de que sea un fichero GPX en concreto. Devuelve una instancia de la clase *DFTrack* con todos los puntos leídos en un dataframe.

3. **readCSV**: se encarga de leer un único fichero CSV y devolver una instancia de la clase *DFTrack*. Cabe decir que leer 1000 ficheros GPX de una media de 150 kB (más de un millón de puntos) tarda alrededor de 3 minutos, mientras que leer exactamente los mismos puntos guardados en un fichero CSV de 136 MB tarda 5 segundos. Esto se debe a que el CSV no tiene que parsear los tags que componen el XML.

3.3.2 Clase *DFTrack*

La base esencial del proyecto es la clase *DFTrack*, ya que es la que permite personalizar las trayectorias importadas o la manera en la queremos que sean las visualizaciones finales. Con ella es posible realizar muchas funciones sobre las trayectorias como, por ejemplo, filtrar, normalizar, concatenar o exportar las trayectorias a CSV.

Esta clase recibe como parámetro de entrada una lista de puntos de trayectorias que, mediante la librería *pandas*, se guardan en un dataframe a partir del cual se han implementado todas las funcionalidades deseadas de una forma muy sencilla y organizada. También usa la librería *geopy* para filtrar las trayectorias por los lugares que se pasan por parámetro.

La librería *pandas* también permite exportar los contenidos de un dataframe en formato CSV o JSON, así que se ha creado una función llamada **export** para tal fin.

Los métodos principales implementados, y que se explican en diferentes capítulos de este documentos, son los siguientes:

1. **getTracksByPlace**: pasándole el nombre de un lugar por parámetro, devuelve todas las trayectorias que pertenezcan a él usando los geocoders Google Geocoding API (V3) o, en caso de error, OpenStreetMap Nominatim.

2. **getTracksByDate** y **getTracksByTime**: basadas en *Time Series* de la librería *pandas*, se encargan de filtrar todos los puntos que pertenezcan a una fecha o un rango de fechas o tiempos.

3. **timeVideoNormalize**: es una de las funciones más importantes de la librería y la que permite que la creación de un vídeo sea más eficiente, ya que se puede analizar más fácilmente lo que se está viendo. Su función es normalizar los puntos de todas las trayectorias importadas en un dataframe para que duren en el vídeo exactamente los segundos indicados por el usuario, es decir, que todos los recorridos empiecen y acaben al mismo tiempo aunque en la realidad no haya sido así. En el capítulo 6 se explica esto con más detalle.

4. **setColors**: a partir de un indicador de las trayectorias, se asigna un color a cada uno de los puntos de las trayectorias.

3.4 Módulo *animation.py*

El módulo *animation.py* es el encargado de sacar las trayectorias procesadas en formato de vídeo, de imagen o de mapa interactivo. Se compone de una única clase llamada *AnimationTrack* y la librería de terceros más importante que utiliza es *matplotlib* para poner cada uno de los puntos que componen las trayectorias en una gráfica. Al instanciar esta clase, se pasa por parámetro una instancia de la clase *DFTrack* con todas las trayectorias, opcionalmente el número de puntos por pulgada que ha de tener la visualización y un booleano indicando si se desea añadir un mapa estático de los lugares por los que pasan las trayectorias como imagen de fondo.

La clase se compone de las siguientes funciones:

1. **computePoints**: es la función que traza punto a punto sobre una figura de *matplotlib* todas las trayectorias, siendo muy útil para cualquier visualización que requiera usar el frame rate como, por ejemplo, los vídeos o la impresión de una imagen en un segundo determinado por el usuario.

2. **computeTracks**: se encarga de trazar las trayectorias completas, por lo que tiene un tiempo computacional muy inferior en comparación con la anterior función, siendo muy útil para exportar imágenes o mapas interactivos de las trayectorias enteras.

3. **makeVideo**: crea un vídeo de las trayectorias.

4. **makeImage**: crea una imagen final con todas las trayectorias pudiéndose especificar, además, que se creen imágenes en uno o varios segundos concretos en los que se desea ver cómo están las trayectorias.

5. **makeMap**: crea un mapa interactivo con las trayectorias procesadas mediante la librería *mplleaflet*.

3.5 Módulo *utils.py*

Dado que hay funciones que solo son usadas por métodos de la librería *Track Animation*, se ha creado este módulo que contiene funciones básicas y concretas para realizar una determinada acción. Estas son:

1. **getPointInTheMiddle**: calcula un nuevo punto entre dos coordenadas dependiendo de una distancia especificada y una dirección. Es útil para normalizar las trayectorias por tiempo.

2. **rgb**: calcula un valor a partir de un mínimo y un máximo para sacar un color en RGB. Es usado para crear el degradado de colores en los puntos según un indicador.

3. **isTimeFormat**: verifica si el valor pasado tiene un formato de tiempo o de fecha. Se utiliza en las funciones para filtrar por rangos de tiempos y fechas para controlar la entrada que dan los usuarios.

Además, este módulo tiene la clase *TrackException*, que se utiliza para lanzar las excepciones que puedan haber en la librería.

CAPÍTULO 4

IMPORTACIÓN DE DATOS GPS A *Track Animation*

La importación de datos GPS a *Track Animation* es una parte importante del proyecto debido a que es necesaria una buena lectura para poder procesar correctamente los datos. El formato principal de almacenamiento de datos GPS que se usa en la librería es el GPX [14], el cual es un estándar basado en XML y fácilmente intercambiable entre distintos dispositivos de GPS, como ya se ha comentado. Para leer este tipo de ficheros y parsearlos, se ha utilizado la librería de Python *gpxpy* [31], cogiendo únicamente los puntos de las trayectorias y poniéndolos en un dataframe de *pandas* [20].

Otro formato desde el que se pueden importar los datos es CSV. Esto se ha hecho porque la librería *pandas* permite exportar e importar este formato a partir de dataframes y su lectura es más rápida que de ficheros GPX, sobre todo si la cantidad de datos a leer es muy grande. Por esta razón, una buena práctica para usar *Track Animation* sería leer inicialmente los ficheros GPX, procesarlos y exportarlos a CSV para que las posteriores lecturas de la misma colección de datos se hagan sobre ese CSV y no se necesite procesarlos de nuevo.

Este capítulo está compuesto de dos secciones que tienen como finalidad explicar cómo se leen los ficheros GPX y CSV en *Track Animation* y cómo se inicializan los dataframes en la clase *DFTrack* con los datos importados.

4.1 Lectura de ficheros en *Track Animation* con *gpxpy*

Para importar datos de GPS de ficheros con formato GPX se ha hecho uso de la librería *gpxpy*, que es capaz de leer dichos ficheros y parsearlos creando diferentes objetos de puntos de interés, rutas, trayectorias y los puntos que las componen. Para ello, se han implementado dos funciones en la clase *ReadTrack* del módulo *tracking.py* [Sección 3.3.1] con las que poder leer tanto un fichero GPX, como un directorio repleto de ellos o un CSV. Cabe recordar que, aunque la librería *gpxpy* permite leer diferentes tipos de puntos, solo se han tenido en cuenta los de las trayectorias.

Al pasarle un fichero de GPX a la librería *gpxpy*, dado que es un XML, primero lo parsea y crea listas con los diferentes objetos mencionados. Es en este momento cuando se hace uso de la función *walk*. Esta devuelve un generador con todos los puntos del fichero GPX para que sea posible recorrerlo e ir guardando la información en una lista para, posteriormente, convertirla en un dataframe en la clase *DFTrack*.

En *Track Animation* se pueden clasificar los datos importados de los ficheros GPX en dos tipos: los que ya vienen escritos en el fichero, como la latitud, la longitud, la elevación o el tiempo, y los que tienen que ser calculados en tiempo de ejecución, como la velocidad o el tiempo y la distancia recorrida entre un punto y otro. Para estos últimos se usan 4 funciones de *gpxpy*: *speed_between*, *time_difference*, *distance_3d* y *distance_2d*. De todos estos campos, hay unos que son esenciales para que *Track Animation* funcione correctamente y puedan usarse todas sus funciones [Cuadro 4.1].

Nombre	**Campo GPX**	**Unidades**	**Descripción**
CodeRoute	-	String	Nombre del fichero GPX
Latitude	lat	Grados decimales (WGS84)	Latitud
Longitude	lon	Grados decimales (WGS84)	Longitud
Date	time	Timestamp	Fecha y hora de creación en formato UTC

Cuadro 4.1: Campos esenciales de un fichero GPX para *Track Animation*

Hay que tener en cuenta que también es posible importar datos de ficheros CSV, por lo que es necesario tener en la cabecera los nombres de los campos tal y como son usados en la librería *Track Animation*, es decir, hay que usar *CodeRoute*, *Latitude*, *Longitude*, etc. Esto se debe a que para leer un CSV se usa la propia librería *pandas*, la cual ya convierte los datos leídos en un dataframe que se pasa, posteriormente, por parámetro a la clase *DFTrack*.

4.2 Conversión de los datos importados a un dataframe de *pandas*

La librería *pandas* es la principal librería para el procesamiento de datos en *Track Animation*. Todos los puntos leídos de las trayectorias se pasan a un dataframe de dicha librería a partir del cual se pueden aplicar distintos tipos de filtros, normalizar los puntos o concatenar varios dataframes de una forma sencilla.

Todo este procesamiento está en la clase *DFTrack* del módulo *tracking.py*. Hay tres opciones para instanciar esta clase:

1. Que no se le pase ninguna lista ni ningún dataframe con puntos, con lo que se crea un dataframe vacío.

Nombre	Campo GPX	Unidades	Descripción
CodeRoute	-	String	Nombre del fichero GPX
Latitude	lat	Grados decimales (WGS84)	Latitud
Longitude	lon	Grados decimales (WGS84)	Longitud
Altitude	ele	Metros	Elevación en metros
Date	time	Timestamp	Fecha y hora de creación en formato UTC
Speed	-	km/h	Velocidad media entre un punto y el siguiente
TimeDifference	-	Segundos	Segundos de diferencia entre un punto y el siguiente
Distance	-	Metros	Distancia recorrida entre un punto y el siguiente

Cuadro 4.2: Campos leídos o calculados de un fichero GPX

2. Que se le pase un dataframe con los puntos de las trayectorias insertados y con los nombres de las columnas especificados en las tablas 4.1 y 4.2.

3. Que se le pase únicamente una lista de puntos, por lo que el dataframe tiene los nombres de columnas especificados en la tabla 4.2. También hay la opción de pasar una lista con los nombres de las columnas que se desee teniendo en cuenta que, como mínimo, deben estar los campos de la tabla 4.1.

El método *__init__* de la clase *DFTrack* es el siguiente:

```
1  def __init__(self, df_points=None, columns=None):
2      if df_points is None:
3          self.df = DataFrame()
4
5      if isinstance(df_points, pd.DataFrame):
6          self.df = df_points
7      else:
8          if columns is None:
9              columns = ['CodeRoute', 'Latitude', 'Longitude', 'Altitude', 'Date',
                   'Speed', 'TimeDifference', 'Distance', 'FileName']
10         self.df = DataFrame(df_points, columns=columns)
```

Algoritmo 4.1: Método *__init__* de la clase *DFTrack*

CAPÍTULO 5

FUNCIONALIDADES BÁSICAS DE LA LIBRERÍA *Track Animation*

Aunque *Track Animation* es una librería que tiene como objetivo la exportación de animaciones de trayectorias de GPS, se han implementado funcionalidades que la complementan para poder hacer un uso más amplio de ella y tener diferentes modos para personalizar dichas animaciones.

Este capítulo va enfocado a explicar tres funcionalidades básicas de la librería *Track Animation* para ayudar a comprender algunos de los algoritmos que se exponen en este documento. Esta funciones son:

- **export**: exportación de trayectorias a CSV y JSON.
- **concat**: concatenación de conjuntos de trayectorias.
- **dropDuplicates**: borrado de puntos duplicados en una trayectoria.

5.1 Exportación de trayectorias a CSV y JSON

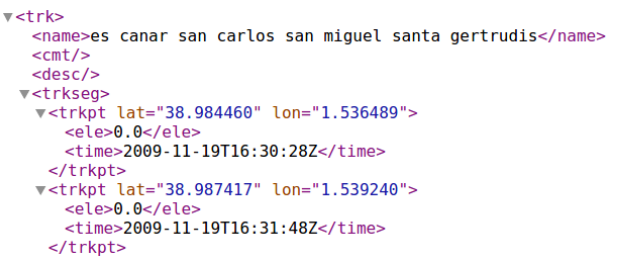
Un dataframe de la librería *pandas* permite ser exportado a diferentes formatos como, por ejemplo, CSV, JSON. En *TrackAnimation* se han utilizado estas funciones uniéndolas en un solo método llamado *export* al que solo se le debe especificar el formato en que se quiere exportar y el nombre del fichero resultante. Es posible también no pasarle ningún parámetro a la función, la cual exporta por defecto a formato CSV bajo el nombre de *'exported_file'*.

```
1 def export(self, export_format='CSV', filename='exported_file'):
2     if export_format == 'JSON':
3         self.df.reset_index().to_json(orient='records', path_or_buf=filename+'.
              json')
4     elif export_format == 'CSV':
```

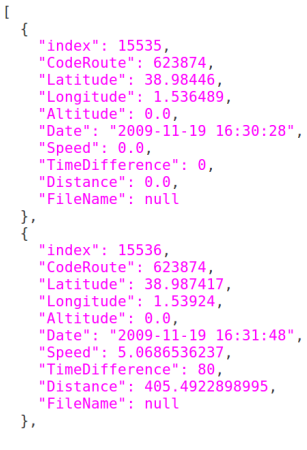
```
5            self.df.to_csv(path_or_buf=filename+'.csv')
6        else:
7            raise TrackException('Must specify a valid format to export', "'%s'" %
                 export_format)
```

Algoritmo 5.1: Exportación de un dataframe de *TrackGPX*

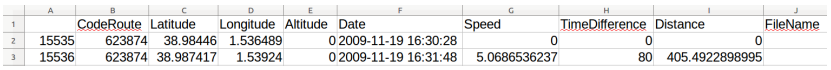
El uso de esta función puede ser muy útil si lo que se desea es crear un conjunto de trayectorias filtrando por fecha o lugar para luego realizar un análisis en una aplicación externa. Otra utilidad es que si se tiene una gran colección de ficheros GPX y se tienen que realizar distintas ejecuciones para una misma colección de datos, puede ser muy tedioso leer cada vez todos los ficheros, por lo que una buena práctica sería realizar una primera ejecución donde se cree el conjunto de datos deseado, exportarlo a CSV y que en cada ejecución posterior se lea dicho CSV. Como ya se ha comentado, leer 1000 ficheros GPX de una media de 150 kB (más de un millón de puntos) tarda alrededor de 3 minutos, mientras que leer exactamente los mismos puntos guardados en un fichero CSV de 136 MB tarda 5 segundos.



(a) Formato GPX



(b) Formato JSON



	CodeRoute	Latitude	Longitude	Altitude	Date	Speed	TimeDifference	Distance	FileName
15535	623874	38.98446	1.536489	0	2009-11-19 16:30:28	0	0	0	
15536	623874	38.987417	1.53924	0	2009-11-19 16:31:48	5.0686536237	80	405.4922898995	

(c) Formato CSV

Figura 5.1: Ejemplo de formato GPX con su equivalencia en formato JSON y CSV

5.2 Concatenación de conjuntos de trayectorias

En *Track Animation* es posible generar varias instancias de la clase *DFTrack* en la que cada una de ellas tiene un dataframe con una colección diferente de datos. Pongamos como ejemplo que de una colección donde se tienen diferentes rutas ciclistas de todas las Islas Baleares, se quiere extraer las que pasan por Palma e Ibiza. Se debe aplicar dos veces el filtrado por lugar: una para Palma y otra para Ibiza. Para generar un vídeo en el que ambos conjuntos salgan sobre el mismo mapa, es necesario unir ambos dataframe y, para ello, se ha creado la función *concat* a la que es necesario pasarle por parámetro

el objeto *DFTrack* que se quiera unir a otro objeto *DFTrack*. Un script de ejemplo para realizar esta tarea es el siguiente:

```
import trackanimation
from trackanimation.animation import AnimationTrack

bi = trackanimation.readTrack('balearic_islands.csv')

palma = bi.getTracksByPlace('Palma')
ibiza = bi.getTracksByPlace('Ibiza')

pal_ibi = palma.concat(ibiza)

fig = AnimationTrack(df_points=pal_ibi, bg_map=True, map_transparency=0.5)
fig.makeVideo(output_file='palma_ibiza')
```

Algoritmo 5.2: Script para concatenar dos *DFTrack*

También es posible pasarle por parámetro a la función *concat* una lista con todos los *DFTrack* que se deseen unir a otro. En el siguiente ejemplo se concatenan Palma, Ibiza y Menorca.

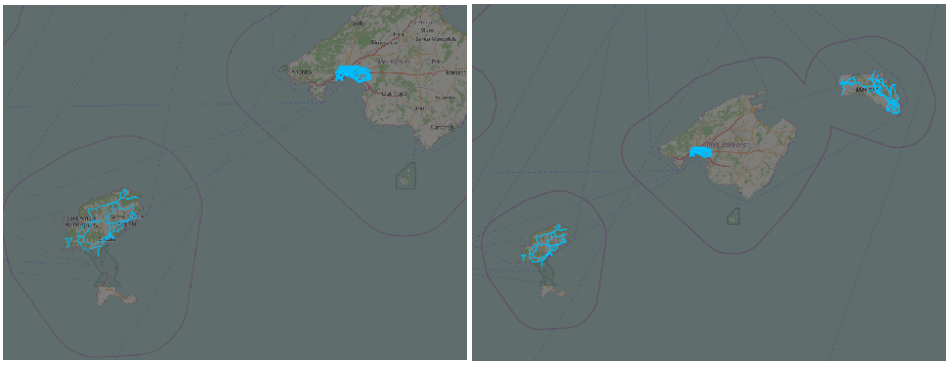
```
import trackanimation
from trackanimation.animation import AnimationTrack

bi = trackanimation.readTrack('balearic_islands.csv')

palma = bi.getTracksByPlace('Palma')
ibiza = bi.getTracksByPlace('Ibiza')
menorca = bi.getTracksByPlace('Menorca')

pal_ibi_men = palma.concat([ibiza, menorca])

fig = AnimationTrack(df_points=pal_ibi_men, bg_map=True, map_transparency=0.5)
fig.makeVideo(output_file='palma_ibiza_menorca')
```

Algoritmo 5.3: Script para concatenar una lista de *DFTrack*

Los resultados de ambos scripts se pueden ver en la figura 5.2.

5.3 Borrado de puntos duplicados en una trayectoria

Es muy posible que un GPS esté programado para que cada x segundos capture un punto. Si un usuario que está realizando una trayectoria se para durante unos minutos y el GPS sigue captando información, son coordenadas repetidas que en algunos casos no pueden ser útiles y lo único que provocan es que el algoritmo sea más lento. Para ello se ha creado la función *dropDuplicates*, la cual borra todos los puntos repetidos en el



(a) Puntos de Palma e Ibiza (b) Puntos de Palma, Ibiza y Menorca

Figura 5.2: Trayectorias concatenadas mediante la función *concat* de *Track Animation*

dataframe a partir de su nombre (*CodeRoute*), latitud (*Latitude*) y longitud (*Longitude)* sin necesidad de pasarle ningún parámetro.

En otro caso que puede ser útil esta función es a la hora de aplicar filtrados por lugares muy próximos entre sí como, por ejemplo, Palma y Marratxí. Los límites de coordenadas de ambos lugares pueden tener cierta desviación que puede provocar que un mismo punto esté en ambos conjuntos.

CAPÍTULO 6

NORMALIZACIÓN DE TRAYECTORIAS DE GPS

El principal problema de los datos guardados de GPS es el muestreo de coordenadas, es decir, la diferencia que hay entre un dispositivo GPS y otro en la precisión y el intervalo de captura de cada punto. Esto puede llevar a que las interpretaciones finales de una o varias trayectorias no sean fiables y no se puedan comparar entre ellas. Además, la calidad o la mala configuración de algunos GPS para capturar los puntos provocaría que la duración de un vídeo exportado con *Track Animation* fuera o muy larga o muy corta, según la cantidad de puntos que tuviera cada trayectoria. Es por este motivo que se ha implementado una función de normalización de los puntos en la que es posible provocar que las trayectorias visualizadas empiecen y acaben al mismo tiempo, independientemente de las horas de inicio y fin de estas. Su principal objetivo es que un usuario pueda controlar la duración de los vídeos y poderlos estudiar más fácilmente.

Esta función, llamada *timeVideoNormalize*, alarga o acorta las trayectorias en base al tiempo total de cada una de ellas y la duración del vídeo especificada por el usuario. Esto provoca que todas acaben y empiecen en el vídeo al mismo tiempo.

Además, se añaden nuevos puntos para dar un efecto de continuidad y uniformidad a las trayectorias al visualizarlas en un vídeo. Para ello se calcula cada cuántos segundos de la duración total de la trayectoria debe tener esta algún punto, independientemente de su número total de puntos. Así pues, si la duración de una trayectoria es de 1200 segundos (20 minutos) y un usuario quiere obtener un vídeo de 30 segundos, la trayectoria debe tener al menos un punto cada 40 segundos de esta (1200s / 30s). En caso de que no haya ningún punto en un margen de 40 segundos, se crea un nuevo punto en proporción a la distancia entre el punto anterior y el punto siguiente más cercano. Por lo tanto, el vídeo generado tiene como mínimo 30 puntos en total repartidos en, al menos, 1 punto por cada segundo de este [Ver figura 6.1]. En el apartado 6.2 se explica la problemática que puede causar la generación de un vídeo con esta cantidad de puntos por segundo.

Se podría haber implementado exactamente el mismo algoritmo utilizando el total de puntos de cada trayectoria. No obstante, debido a que los intervalos entre puntos para una misma trayectoria pueden ser diferentes, la visualización de estos no

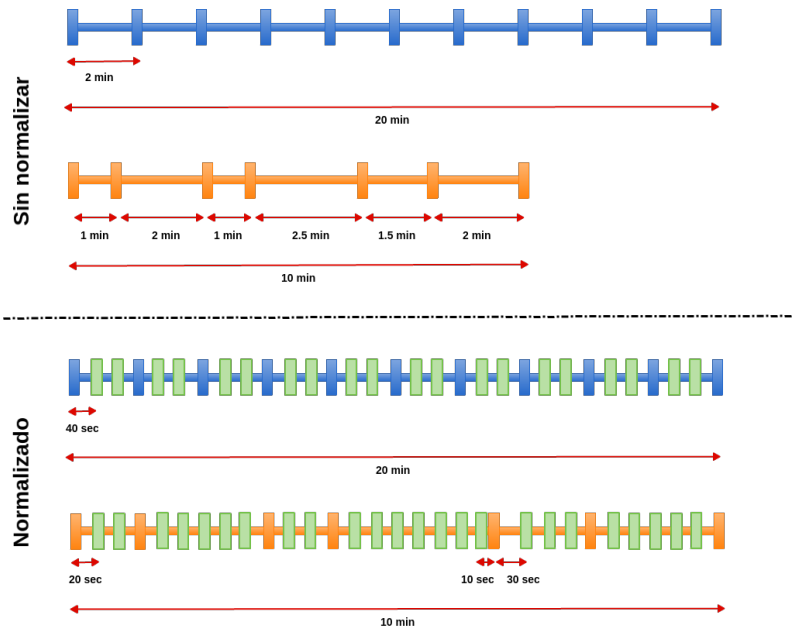


Figura 6.1: Comparación de dos trayectorias sin normalizar y normalizadas

es uniforme y hay mayores saltos cuanto más largo sea el intervalo y cuanta menor cantidad de puntos haya en una trayectoria. Además, habría la misma cantidad de puntos tanto en un intervalo muy pequeño como en uno muy grande. En cambio, dado que el algoritmo implementado solo tiene en cuenta el tiempo total de cada trayectoria, los puntos quedan uniformes y prácticamente a la misma distancia entre ellos ya que asegura que cada x segundos siempre haya algún punto.

Este capítulo se divide en tres secciones. La primera de ellas explica los conceptos básicos para entender lo que hace el algoritmo de normalización. En la segunda sección se muestra la importancia que tienen los fotogramas por segundo en la normalización para crear un efecto de uniformidad y continuidad a las trayectorias. El último apartado explica cómo se crean nuevos puntos realizando algunos cálculos para conocer sus coordenadas.

6.1 Conceptos básicos del algoritmo de normalización

Uno de los objetivos de la normalización a partir del tiempo total de cada trayectoria es ver un vídeo en el que todas las trayectorias, independientemente de la duración que tengan o de su número total de puntos, empiecen y acaben al mismo tiempo. Para ello, lo que se ha hecho es alargar o acortar cada trayectoria en base al tiempo en el que se desea que se vea el vídeo.

El algoritmo de normalización recorre todas las trayectorias calculando, por cada una de ellas, cada cuántos segundos debe tener al menos un punto y asigna el fotograma en el que debe ir cada uno de los puntos en la visualización en una nueva columna del dataframe, llamada *VideoFrame*. Además, para solucionar el problema de los diferentes intervalos de cada trayectoria, el algoritmo es capaz de insertar tantos puntos nuevos entre el punto anterior y el siguiente en el intervalo como sean necesarios, minimizando

así la longitud entre estos y haciendo más uniforme la trayectoria. Esto se explica en profundidad en el apartado 6.3. Hay que tener dos aspectos en cuenta para que esta función tenga los efectos deseados:

1. El dataframe ha de tener un campo llamado *TimeDifference*, que es la diferencia en segundos entre un punto y otro. Este campo se utiliza para extraer el tiempo acumulado desde que se inicia la trayectoria hasta cualquier punto de esta y usarlo como "índice"para poder obviar el total de puntos de la trayectoria y basarnos únicamente en la duración de esta.

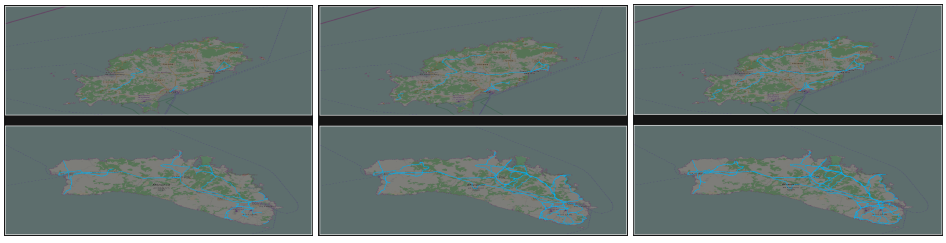
2. A la función de normalización hay que pasarle también por parámetro los fotogramas por segundos que se desea que tenga el vídeo. Esto provoca que se reduzca el tiempo de diferencia en el que debe haber al menos un punto en la trayectoria, añadiéndose más puntos intermedios en caso de que sea necesario y reduciéndose aún más los intervalos entre ellos. De esta forma se generan más imágenes que, al unirlas para formar el vídeo, engañan al ojo humano creando un efecto de continuidad en las trayectorias. Esto se explica con más detalle en la sección 6.2.

Una de las funcionalidades que se le puede dar a este método es que se puede normalizar un grupo de trayectorias en base a un tiempo deseado y luego normalizar otro conjunto de trayectorias diferentes en función de otro tiempo. Como resultado se puede exportar un vídeo que compare ambos conjuntos normalizados, siendo la duración del vídeo igual al tiempo del conjunto más largo. No obstante, hay que tener en cuenta que los fotogramas por segundo para ambas normalizaciones ha de ser el mismo, ya que sino el vídeo no tendría la duración deseada. En la figura 6.2 se puede ver una comparación de dos grupos de trayectorias donde las de Menorca acabaron a los 10 segundos, pero el vídeo continuó hasta los 30 porque se especificó que se normalizaran las trayectorias de Ibiza para que duraran ese tiempo.

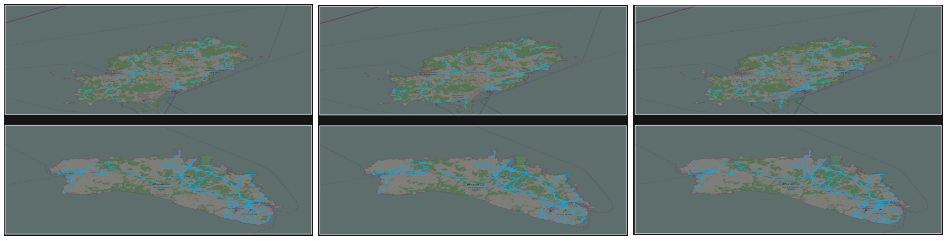
Lo mismo ocurriría si concatenamos ambos conjuntos mediante la función *concat* de *Track Animation* y exportáramos el dataframe resultante. La diferencia sería que, en lugar de verlo en dos gráficas diferentes, lo veríamos en una.

6.2 Importancia de los fotogramas por segundo en la normalización

Tanto en el mundo de los videojuegos como en el cinematográfico, las imágenes por segundo (*Frames Per Second (FPS)* o *frame rate*) han tenido una gran relevancia. Los FPS es la tasa o velocidad a la que se reproducen las imágenes (fotogramas) de un vídeo. Varios estudios han demostrado que cuanto mayor son los FPS, mayor es la jugabilidad, la precisión y la calidad en el movimiento de objetos [41, 42, 43]. Por ejemplo, en [41] se demostró como un frame rate de 60 FPS dio un 14 % más de resultados mejores que uno de 30, pero sin grandes diferencias comparándolo con uno de 45 FPS. También, en [42] se pudo demostrar que el público prefería, con grandes creces, un frame rate de 60 sobre uno de 24 al ver películas. No obstante, había discrepancias entre los 48 FPS y los 60 FPS entre diferentes tipos de reproducción y diferentes movimientos.



(a) 5 segundos (b) 10 segundos (c) 15 segundos



(d) 20 segundos (e) 25 segundos (f) 30 segundos

Figura 6.2: Secuencia de imágenes de un vídeo comparando trayectorias ciclistas de Ibiza y Menorca

Los fotogramas por segundo se han tenido en cuenta a la hora de implementar *Track Animation*, dando la posibilidad a un usuario de elegir con qué tasa de fotogramas quiere exportar su vídeo. Esto se ha hecho debido a las diferencias en las que cada punto de las trayectorias es capturado. Si no se hubiera tenido en cuenta el frame rate, en cada segundo de un vídeo se vería un salto entre un punto y otro, y más aún cuanto más intervalo de captura haya entre ellos. Mediante el frame rate, lo que se provoca es que el margen de segundos en el que una trayectoria debería tener al menos un punto sea más pequeño, creándose así más puntos intermedios nuevos. Esto da un mayor efecto de continuidad en la visualización de cada trayectoria.

Así pues, continuando con el ejemplo del apartado introductorio de este capítulo, si la duración de una trayectoria es de 1200 segundos y un usuario quiere obtener un vídeo de 30 segundos con un frame rate de 20, el cálculo que realiza el algoritmo es

$$time\_diff = 1200s/30s/20fps = 2$$

por lo que la trayectoria debe tener al menos un punto cada 2 segundos de esta. Si no es así, se crea un nuevo punto. Dicho de otra manera, el vídeo generado tendrá 600 fotogramas y, por lo tanto, 600 puntos como mínimo. Estos son repartidos en, al menos, 20 puntos por cada segundo de reproducción, es decir, un punto por cada fotograma, lo que provoca en el ojo humano un efecto de continuidad en la trayectoria aunque sus intervalos de captura sean muy grandes o distintos. Hay que tener en cuenta que si *time_diff* es mayor que el margen en el que el GPS ha capturado los puntos, dentro de un mismo fotograma habrá varios de ellos. Por este motivo se han usado las palabras "al menos" o "como mínimo" al especificar el número de fotogramas o puntos por segundo que tendrá el vídeo.

Es necesario que, si se desea mantener la duración especificada del vídeo, se indique el mismo frame rate en la normalización y en la exportación de este. Lo mismo ocurre

si se quiere normalizar más de un grupo de trayectorias para ponerlas en el mismo vídeo.

6.3 Creación de nuevos puntos intermedios

El algoritmo de normalización añade nuevos puntos intermedios entre otros dos cuando los intervalos son más largos que el tiempo calculado en la normalización. Dado que cada intervalo puede ser diferente el uno del otro, para añadir un nuevo punto no se tiene en cuenta la longitud ni la duración de la trayectoria completa, sino la de cada uno de los intervalos de esta. De esta forma se consigue que el nuevo punto quede proporcional al tiempo y a la longitud del intervalo y, por lo tanto, a la trayectoria en sí. Para ello, se calcula el tiempo proporcional en el que debe de ir el nuevo punto mediante la fórmula:

$$time\_proportion(point\_idx) = \frac{time\_diff \times point\_idx}{end\_point['TimeDifference']}$$

donde *time_diff* es la diferencia de tiempo en el que la trayectoria debería tener al menos un punto, *point_idx* es el total de número de puntos nuevos que ya se han creado en ese intervalo y *end_point['TimeDifference']* indica el número de segundos total que tiene el intervalo.

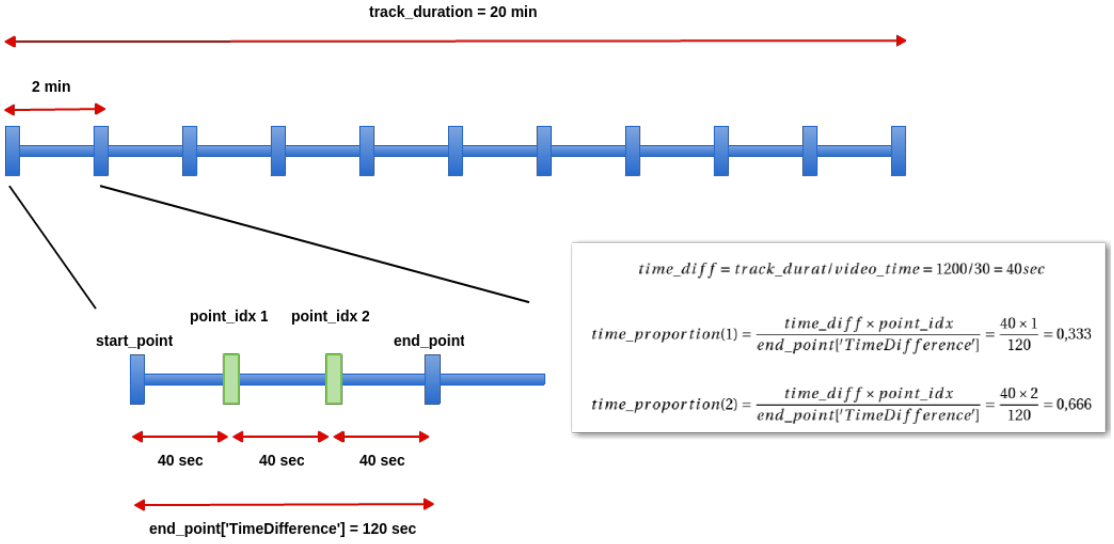


Figura 6.3: Adición de puntos nuevos a una trayectoria

Posteriormente, se calcula la distancia y el tiempo proporcional en el que debe ir el nuevo punto en el intervalo. Con ello ya es posible calcular la velocidad relativa a la que el sujeto iba en ese momento mientras hacía la trayectoria, la hora y la altitud relativa. Por último, usando un objeto *Point* de la librería *geopy* con las coordenadas del punto inicial del intervalo y otro con las del punto final, se calculan las coordenadas del nuevo punto especificando a qué distancia está respecto al punto inicial. El algoritmo simplificado es el siguiente:

```
def getPointInTheMiddle(start_point, end_point, time_diff, point_idx)):
    time_proportion = (time_diff * point_idx) / end_point['TimeDifference']

    distance_proportion = end_point['Distance'] * time_proportion
    time_diff_proportion = end_point['TimeDifference'] * time_proportion
    speed = distance_proportion / time_diff_proportion
    cum_time_diff = start_point['CumTimeDiff'] + time_diff_proportion
    date = start_point['Date'] + time_diff_proportion
    altitude = (end_point['Altitude'] + start_point['Altitude']) / 2

    geo_start = geopy.Point(start_point['Latitude'], start_point['Longitude'])
    geo_end = geopy.Point(end_point['Latitude'], end_point['Longitude'])
    middlePoint = getCoordinates(geo_start, geo_end, distance_proportion)

    df_middlePoint = ([[middlePoint.latitude, middlePoint.longitude, altitude,
        date, speed, cum_time_diff]])

    return df_middlePoint
```

Algoritmo 6.1: Creación de un nuevo punto intermedio

Para calcular las coordenadas de un nuevo punto, es necesario conocer la distancia de este al punto inicial y la dirección en grados en la que va la trayectoria. Para ello, se ha utilizado el algoritmo de [44]. Cabe decir que, aunque se instancia la clase *VincentyDistance* de *geopy*, no se ha usado la fórmula de Vincenty [45] para calcular la distancia entre el punto inicial y el nuevo ya que, como se ha visto en el algoritmo anterior, esta se calcula mediante la distancia proporcional del primer punto del intervalo y el último. Únicamente se usa la clase *VincentyDistance* para pasar por parámetro la distancia ya calculada. Posteriormente, un algoritmo de la librería *geopy* calcula las nuevas coordenadas.

```
def getCoordinates(start_point, end_point, distance_meters):
    # ********************************************
    # Get bearing
    # ********************************************
    start_lat = math.radians(start_point.latitude)
    start_lng = math.radians(start_point.longitude)
    end_lat = math.radians(end_point.latitude)
    end_lng = math.radians(end_point.longitude)

    d_lng = end_lng - start_lng
    if abs(d_lng) > math.pi:
        if d_lng > 0.0:
            d_lng = -(2.0 * math.pi - d_lng)
        else:
            d_lng = (2.0 * math.pi + d_lng)

```

```
17      tan_start = math.tan(start_lat / 2.0 + math.pi / 4.0)
18      tan_end = math.tan(end_lat / 2.0 + math.pi / 4.0)
19      dPhi = math.log(tan_end / tan_start)
20      bearing = (math.degrees(math.atan2(d_lng, dPhi)) + 360.0) % 360.0;
21
22      # ******************************************
23      # Get coordinates
24      # ******************************************
25      distance_km = distance_meters / 1000
26      d = geo_dist.VincentyDistance(kilometers=distance_km)
27      destination = d.destination(point=start_point, bearing=bearing)
28
29      return geopy.Point(destination.latitude, destination.longitude)
```

Algoritmo 6.2: Cálculo de las coordenadas para un nuevo punto intermedio

CAPÍTULO 7

FILTRADO DE TRAYECTORIAS DE GPS EN *Track Animation*

Intentar interpretar la evolución del movimiento de una colección de trayectorias muy grande puede ser una tarea compleja. Es por eso que una de las características esenciales para realizar un análisis en cualquier área es la creación de pequeños subconjuntos de datos a partir de un conjunto más grande. Para obtener estos subconjuntos es necesario filtrar los datos y procesarlos hasta obtener la información deseada. Una trayectoria de GPS está compuesta por puntos de coordenadas que tienen asociada una fecha, por lo que poder aplicar filtros a estos valores es algo esencial.

En *Track Animation* se han creado tres funciones con las que poder filtrar las trayectorias por lugar, fecha y tiempo. Estos criterios permiten analizar y comparar datos más concretos, pudiendo aplicar diferentes indicadores y normalizaciones para su visualización final. Estas funciones son *getTracksByPlace*, *getTracksByDate* y *getTracksByTime*, respectivamente. La primera de ellas utiliza los geocoders de *GoogleV3* y *OpenStreetMap* de la librería *geopy* para devolver los puntos que hay en un lugar en concreto o los de todas las trayectorias que hayan pasado alguna vez por ese lugar. El filtrado por fecha y por tiempo utiliza las herramientas para series temporales de la librería *pandas*.

Con estas tres funciones es posible, por ejemplo, sacar un conjunto de datos con los puntos de Palma que hay entre las 09:00 y las 12:00 del primer cuatrimestre del año desde el 01/01/2015 hasta el 01/06/2017. El resultado de aplicar estos filtros se puede ver en la figura 7.1.

```
1 import trackanimation
2 from trackanimation.tracking import DFTrack
3 from trackanimation.animation import AnimationTrack
4
5 bi = trackanimation.readTrack('balearic_islands.csv')
6
7 valldemossa = bi.getTracksByPlace('Valldemossa', only_points=False)
8 vall_time = valldemossa.getTracksByTime('08:00', '12:00')
```

```
vall_2010 = vall_time.getTracksByDate(start='2010-01-01', end='2010-04-30')
vall_2011 = vall_time.getTracksByDate(start='2011-01-01', end='2011-04-30')
vall_2012 = vall_time.getTracksByDate(start='2012-01-01', end='2012-04-30')
vall_2013 = vall_time.getTracksByDate(start='2013-01-01', end='2013-04-30')
vall_2014 = vall_time.getTracksByDate(start='2014-01-01', end='2014-04-30')

vall_filter = vall_2010.concat([vall_2011, vall_2012, vall_2013, vall_2014])

fig = AnimationTrack(df_points=vall_filter, bg_map=True, map_transparency=0.5)
fig.makeVideo(output_file='valldemossa_filtered')
```

Algoritmo 7.1: Script de filtrado de trayectorias por lugar, fecha y tiempo

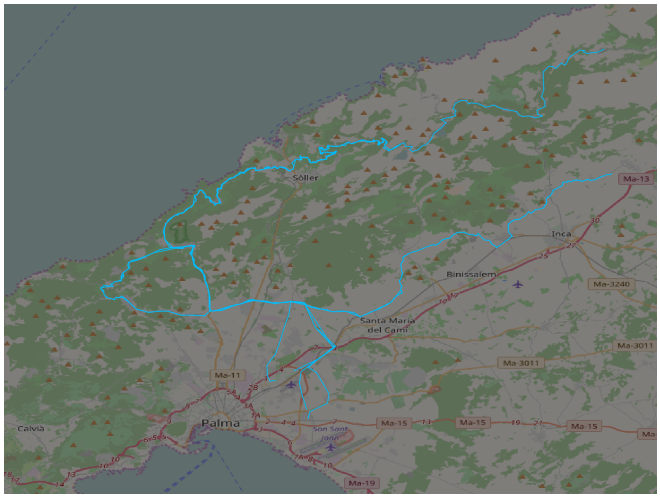


Figura 7.1: Trayectorias que pasan por Valldemossa entre las 08:00 y las 12:00 del primer cuatrimestre del año desde el 01/01/2010 hasta el 01/06/2014

Este capítulo está compuesto de dos secciones donde se detalla cómo se ha implementado y cómo funciona el filtrado por lugar y el filtrado por fecha y tiempo, respectivamente.

7.1 Filtrado por lugar

La gran cantidad de trayectorias mostradas en un vídeo hace difícil llegar a interpretar correctamente los datos. Es por eso que se hace esencial analizar conjuntos de datos más pequeños y centrados en lugares concretos. Esto permite que también se puedan comparar varios lugares o distintos indicadores para un mismo lugar. La función creada se llama *getTracksByPlace* y se encuentra en la clase *DFTrack*. Dicha función llama a otras dos funciones, *getTracksByPlaceGoogle* y *getTracksByPlaceOSM*, que son las encargadas de parsear el JSON que devuelven las API's para extraer los límites de un lugar mediante los servicios de Google o los de Open Street Map, respectivamente, usando los geocoders de la librería *geopy*. El funcionamiento del algoritmo es llamar primero

a la API de Google y, en caso de que falle, dé un *timeout* o no devuelva información, llame a la de Open Street Map.

A las tres funciones es posible introducirle los siguientes tres parámetros:

- **place**: el lugar del que se quiere recuperar los puntos.
- **timeout**: el tiempo de espera del servicio del geocoder para responder antes de devolver un valor nulo.
- **only_points**: booleano en el que se especifica si lo que se desea recuperar son únicamente los puntos que pasan por un lugar o las trayectorias completas que han pasado por ese lugar.

El parámetro *only_poins* es realmente importante por el efecto que puede causar en una visualización. Si se crea un conjunto especificando que solo se desean los puntos que pasan por un determinado lugar y luego se concatena con otro conjunto de otro lugar, es posible que haya puntos de una misma trayectoria en ambos conjuntos, lo que significa que quien hizo esa trayectoria pasó por ambos lugares. En la visualización final esto puede provocar que haya líneas totalmente rectas entre un lugar y otro debido a que los puntos intermedios se han obviado en el filtrado. Pongamos el caso de que deseamos obtener los puntos que pasan por Sóller y los que pasan por Manacor. El resultado final es el que se puede apreciar en la figura 7.2.

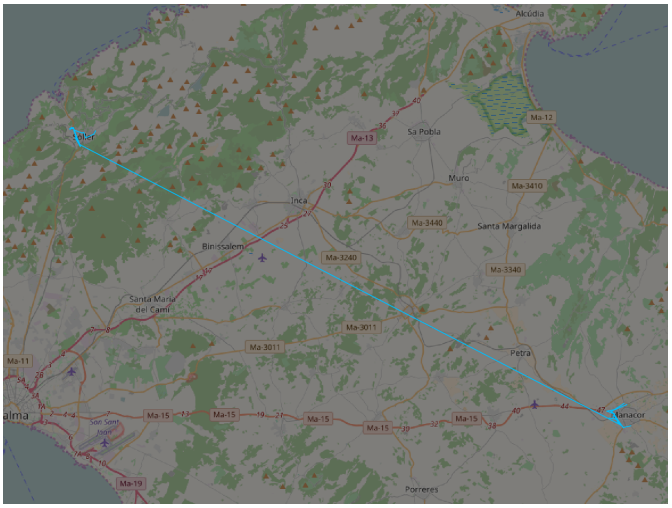


Figura 7.2: Puntos de Sóller y Manacor

En el caso en que el parámetro *only_points* se ponga a falso, en la visualización final se podrán ver todas las trayectorias que hayan pasado por un determinado lugar [Figura 7.3].

A partir de esta función, es posible procesar los datos de distintas maneras con el fin de obtener otros resultados. Como ya se ha comentado, el objeto principal de la clase *DFTrack* es un dataframe que contiene puntos de trayectorias. Este dataframe puede ser usado fuera de la propia librería *Track Animation* utilizando las características y herramientas de la librería *pandas*. Un ejemplo es el de poder obtener únicamente las trayectorias que han pasado por varios lugares, como se muestra en el siguiente script,

Figura 7.3: Trayectorias que han pasado por Manacor

donde el resultado final [Figura 7.4] muestra los puntos de Sóller y Manacor más las trayectorias que han pasado por ambos lugares:

```
import trackanimation
from trackanimation.tracking import DFTrack
from trackanimation.animation import AnimationTrack

bi = trackanimation.readTrack('balearic_islands.csv')

# Get all the tracks that cross Soller and Manacor, separately
soller_trk = bi.getTracksByPlace('Soller', only_points=False)
manacor_trk = bi.getTracksByPlace('Manacor', only_points=False)

# Get their dataframes
soller_df = soller_trk.df
manacor_df = manacor_trk.df

# Get the tracks that cross Soller and Manacor
soller_list = soller_df['CodeRoute'].unique().tolist()
sol_man_df = manacor_df[manacor_df['CodeRoute'].isin(soller_list)]
sol_man_trk = DFTrack(sol_man_df)

# Get only the points of Soller and Manacor, separately
soller_pnt = bi.getTracksByPlace('Soller', only_points=True)
manacor_pnt = bi.getTracksByPlace('Manacor', only_points=True)

# Concatenate points of Soller and Manacor with the tracks
# that cross the two places
sol_man_trk = sol_man_trk.concat([soller_pnt, manacor_pnt])

```

```
fig = AnimationTrack(df_points=sol_man_trk, bg_map=True, map_transparency=0.5)
fig.makeVideo(output_file='soller_manacor')
```

Algoritmo 7.2: Script para visualizar los puntos de Sóller y Manacor más las trayectorias que han pasado por ambos lugares

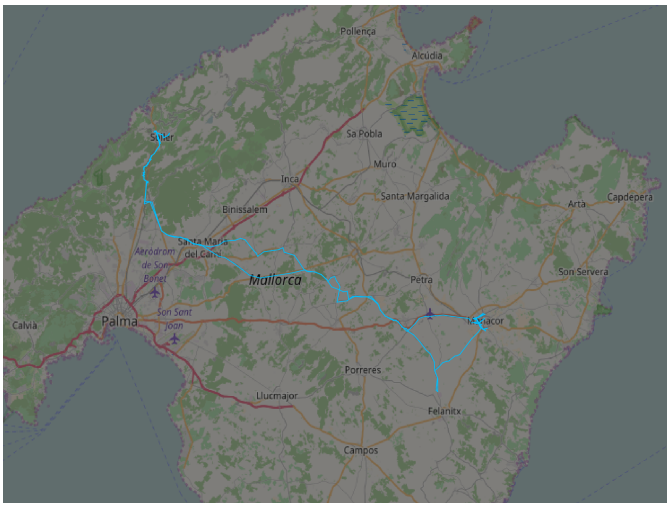


Figura 7.4: Puntos de Sóller y Manacor más las trayectorias que han pasado por ambos lugares

7.2 Filtrado por fecha y tiempo

Se ha implementado una forma simple de filtrar por fecha o tiempo para acotar aún más los datos y analizar las trayectorias en base a periodos de tiempos. Para ello se utilizan las herramientas para series temporales de la librería *pandas*, que no es más que una forma estructurada de datos temporales que indican un instante específico en el tiempo (timestamp), periodos fijos como un mes o un año completos o intervalos de tiempo desde un timestamp inicial a otro final [2].

Las series temporales de *pandas* se usan para que, a partir de un conjunto de datos ya creado, se añada el rango de fechas establecido como índice del conjunto para particionar, agregar, agrupar, etc. los datos a partir de dicho índice. Sin embargo, lo que se desea hacer en *Track Animation* es que a partir de un conjunto de trayectorias, se seleccionen las que pertenecen a un periodo o intervalo de fechas o tiempos. Dicho de otra forma, las herramientas de series temporales de *pandas* se han usado en *Track Animation* únicamente para crear rangos de fechas y no para crear conjuntos de datos con series temporales. De este modo nos aseguramos de que la estructura del dataframe no cambia y, además, es posible que en una misma función implementada se puedan filtrar las trayectorias por cualquiera de los tres grupos temporales comentados en el párrafo anterior.

Para ello, se han creado dos funciones: *getTracksByDate* para filtrar por un intervalo o periodos de fecha y *getTracksByTime* para hacerlo a partir de un rango de tiempos.

7.2.1 Función *getTracksByDate*

Esta función es la encargada de filtrar las trayectorias por un rango o un intervalo de fechas a partir de cuatro parámetros:

- **start**: fecha inicial del periodo.
- **end**: fecha final del periodo.
- **periods**: número de periodos desde la fecha inicial o hasta la fecha final.
- **freq**: frecuencia con la que se coge un nuevo periodo.

Solo es posible especificar dos de los tres parámetros *start*, *end* o *periods*. El parámetro *freq* tiene por defecto el valor 'D', que quiere decir que los periodos son diarios. Existen otros valores que pueden ser usados en este parámetro y que están especificados en el cuadro 7.1.

Alias	**Tipo**	**Descripción**
D	Day	Calendario diario
B	BusinessDay	Calendario diario laboral
M	MonthEnd	Último día del mes del calendario
BM	BusinessMonthEnd	Último día del mes del calendario laboral
MS	MonthBegin	Primer día del mes del calendario
BMS	BusinessMonthBegin	Primer día del mes del calendario laboral
W-MON, W-TUE...	Week	Fechas semanales en un día específico de la semana
WOM-1MON, WOM-2MON...	WeekOfMonth	Genera semanalmente fechas en la primera, segunda, tercara o cuarta semana del mes
Q-JAN, Q-FEB...	QuarterEnd	Fechas cuatrimestrales del último día del calendario de cada mes, acabando en el mes especificado
BQ-JAN, BQ-FEB...	BusinessQuarterEnd	Fechas cuatrimestrales del último día del calendario laboral de cada mes, acabando en el mes especificado

QS-JAN, QS-FEB...	QuarterBegin	Fechas cuatrimestrales del primer día del calendario de cada mes, acabando en el mes especificado
BQS-JAN, BQS-FEB...	BusinessQuarterBegin	Fechas cuatrimestrales del primer día del calendario laboral de cada mes, acabando en el mes especificado
A-JAN, A-FEB...	YearEnd	Fechas anuales del último día del calendario del mes especificado
BA-JAN, BA-FEB...	BusinessYearEnd	Fechas anuales del último día del calendario laboral del mes especificado
AS-JAN, AS-FEB...	YearBegin	Fechas anuales del primer día del calendario del mes especificado
BAS-JAN, BAS-FEB...	BusinessYearBegin	Fechas anuales del primer día del calendario laboral del mes especificado

Cuadro 7.1: Frecuencias admitidas por las series temporales de *pandas* [2]

La función *getTracksByDate* utiliza el método de la librería *pandas date_range* al que se le pasan exactamente los mismos parámetros. Esta función retorna únicamente unas fechas concretas (con formato *YYYY-MM-DD*) según las fechas, periodos o frecuencias especificadas. Posteriormente, los valores de las fechas de los puntos de las trayectorias de GPS, que tienen un formato de fecha UTC (*YYYY-MM-DDThh:mm:ssz*), se convierten a formato *YYYY-MM-DD* para hacerlas coincidir con las fechas devueltas anteriormente por la función *date_range*. El algoritmo es el siguiente:

```
def getTracksByDate(self, start=None, end=None, periods=None, freq='D'):
    rng = pd.date_range(start=start, end=end, periods=periods, freq=freq)

    df['Date'] = pd.to_datetime(df['Date'])
    df['ShortDate'] = df['Date'].apply(lambda x: x.date().strftime('%Y-%m-%d'))
    df = df[df['ShortDate'].apply(lambda date: date in rng)]
    del df['ShortDate']

    return self.__class__(df, list(df))
```

Algoritmo 7.3: Función *getTracksByDate* de *Track Animation*

Algunos ejemplos de uso de esta función son los siguientes:

```
"""
DatetimeIndex(['2010-01-01', '2010-01-02', '2010-01-03', '2010-01-04'
               ...
               '2012-04-27', '2012-04-28', '2012-04-29', '2012-04-30'],
              dtype='datetime64[ns]', length=851, freq='D')
"""
tracks.getTracksByDate(start='2010-01-01', end='2012-04-30')

"""
DatetimeIndex(['2010-01-01', '2010-01-02', '2010-01-03', '2010-01-04',
               ...
               '2010-01-07', '2010-01-08', '2010-01-09', '2010-01-10'],
              dtype='datetime64[ns]', freq='D')
"""
tracks.getTracksByDate(start='2010-01-01', periods=10)

"""
DatetimeIndex(['2010-01-04', '2010-01-11', '2010-01-18', '2010-01-25',
               ...
               '2012-04-09', '2012-04-16', '2012-04-23', '2012-04-30'],
              dtype='datetime64[ns]', length=122, freq='W-MON')
"""
tracks.getTracksByDate(start='2010-01-01', end='2012-04-30', freq='W-MON')

"""
DatetimeIndex(['2010-02-28', '2010-05-31', '2010-08-31', '2010-11-30',
               ...
               '2011-08-31', '2011-11-30', '2012-02-29', '2012-05-31'],
              dtype='datetime64[ns]', freq='Q-MAY')
"""
tracks.getTracksByDate(start='2010-01-01', periods=10, freq='Q-MAY')

"""
DatetimeIndex(['2003-01-31', '2004-01-31', '2005-01-31', '2006-01-31',
               ...
               '2009-01-31', '2010-01-31', '2011-01-31', '2012-01-31'],
              dtype='datetime64[ns]', freq='A-JAN')
"""
tracks.getTracksByDate(end='2012-04-30', periods=10, freq='A-JAN')
```

Algoritmo 7.4: Ejemplos de uso de la función *getTracksByDate*

7.2.2 Función *getTracksByTime*

Esta función se ha implementado para que, a parte de filtrar por fechas, también se pueda filtrar por intervalos de tiempo. Para ello se ha usado la función *indexer_between_time* de la librería *pandas* a la que se le pasa una hora inicial y una hora final. También es posible indicarle si ambas horas deben ir incluidas en el filtro, estando activado por defecto.

Para esta función ha sido necesario comprobar si los formatos de horas pasados por parámetro eran correctos. Estos pueden ser:

```
1 TIME_FORMATS = ['%H:%M', '%H%M', '%I:%M%p', '%I%M%p', '%H:%M:%S', '%H%M%S', '%I
      :%M:%S%p', '%I%M%S%p']
```

Algoritmo 7.5: Formatos de horas aceptados para filtrar en *Track Animation*

siendo *H* e *I* las horas en formato de 24 y 12 horas, respectivamente, *M* los minutos, *S* los segundos y *p* para indicar si es AM o PM.

Una vez comprobados los formatos, se ha creado una variable del tipo *DatetimeIndex* con los valores de la columna de fecha de las trayectorias para poder hacer uso de la función *indexer_between_time*. Los valores devueltos se filtran en el dataframe mediante la función *iloc*, que no es más que un selector por posición usando booleanos.

CAPÍTULO 8

VISUALIZACIÓN DE INDICADORES A PARTIR DE COLORES SOBRE LAS TRAYECTORIAS

Los indicadores son esenciales para poder analizar e interpretar grandes cantidades de datos de una forma eficaz y sencilla. Dado que *Track Animation* únicamente exporta visualizaciones de las trayectorias, se ha implementado la función *setColors* para poder ver los valores de los indicadores aplicando una paleta de colores a las trayectorias. Altitud, velocidad, tiempo recorrido o cualquier otro indicador personalizado que cree un usuario puede convertirse en un degradado que se hace visible cuando se exporta un vídeo, una imagen o un mapa interactivo.

Los indicadores deben tener un valor por cada punto de la trayectoria. El color que se aplica a cada punto se calcula en base al valor máximo y mínimo del indicador. Así pues, en la tabla 8.1 se puede ver un ejemplo de cuáles son los códigos de los colores en RGB que se le asigna a un indicador que va del 1 al 3. Como se puede observar, los valores de RGB van del 0 al 255. No obstante, la librería *matplotlib* solo admite valores entre el 0 y el 1 para pintarlos sobre una figura.

Valor	R	G	B
1.0	0	0	255
1.2	0	51	204
1.4	0	102	153
1.6	0	153	102
1.8	0	204	51
2.0	0	255	0
2.2	50	205	0

2.4	101	154	0
2.6	153	102	0
2.8	204	52	0
3.0	255	0	0

Cuadro 8.1: Ejemplo de asignación de colores a un indicador

La paleta de colores utilizada para cualquier indicador es la que se muestra en la figura 8.1, siendo el azul el valor más bajo del indicador y el rojo el valor más alto.

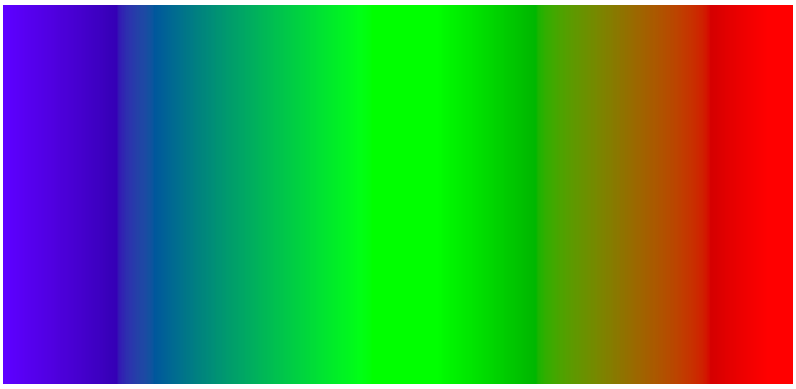


Figura 8.1: Paleta de colores usada por los indicadores de las trayectorias

Para que un usuario pueda personalizar más las visualizaciones y comparar entre trayectorias, es posible especificar si se desea que se aplique la paleta de colores a cada trayectoria individualmente o a todas las trayectorias en conjunto. Esto se indica mediante el parámetro *individual_tracks* de la función *setColors*, el cual está por defecto a verdadero. Un ejemplo de ello se puede ver en la figura 8.2. Como se puede observar, la imagen a) tiene colores que van del azul al rojo en cada una de las trayectorias, pudiéndose ver en qué tramos de cada trayectoria se ha ido más rápido o más lento. No obstante, en la imagen b) se aplican los colores al conjunto de todas las trayectorias, haciendo posible ver cuál de ellas ha sido la más veloz, por ejemplo.

(a) Por trayectoria

(b) Trayectorias en conjunto

Figura 8.2: Comparación del degradado de colores del indicador de velocidad aplicado a cada trayectoria individualmente y en conjunto

CAPÍTULO 9

VISUALIZACIÓN DE LAS TRAYECTORIAS PROCESADAS EN *Track Animation*

El objetivo principal de *Track Animation* es generar visualizaciones de trayectorias de GPS almacenadas en ficheros con formato GPX. Esta parte es la que provoca que las trayectorias puedan ser fácilmente interpretables a partir del manejo que se ha hecho de los datos, ya sea especificando los segundos de duración o el número de FPS de un vídeo, sobre qué indicador se han de crear los colores, qué lugares se han de ver o qué puntos han de ser descartados.

Todo este procesamiento se puede ver plasmado en tres tipos de visualizaciones diferentes: *vídeos*, *imágenes* y *mapas interactivos* de OpenStreetMap. Para ello se hace uso de la librería *matplotlib* que genera una figura de una o varias gráficas con todos los puntos de uno o varios dataframes de la clase *DFTrack*. Se ha diseñado un algoritmo que gestiona la forma en la que los puntos de las diferentes trayectorias se visualizan con el objetivo de que cada una de ellas sea independiente de la otra y generen el mismo recorrido que se hizo en la realidad. No obstante, para grandes cantidades de datos y para visualizaciones estáticas o que no requieran de fotogramas, como podría ser un mapa interactivo, se ha diseñado otro algoritmo que plasma en la figura las trayectorias completas, sin ir punto a punto.

FFmpeg [19] es la herramienta que se usa para generar los vídeos a partir de secuencias de imágenes creadas por *matplotlib* en base al número de FPS especificado por el usuario. Cuanto mayor sea el número de FPS, más rápida es la visualización de trayectorias en caso de que no haya habido una normalización anterior de los puntos que las componen [Capítulo 6]. Se ha usado la librería *mplleaflet* [38] para facilitar la visualización de las trayectorias en un mapa interactivo de OpenStreetMap [Figura 9.1].

Para la implementación de todo lo relacionado con las visualizaciones, se ha creado la clase *AnimationTrack* dentro del módulo *animation.py*. El objeto principal de dicha clase es la figura creada con *matplotlib*, la cual tiene una gráfica por cada *DFTrack* pasado por parámetro. Sobre estas gráficas se van añadiendo los puntos para, finalmente, generar uno de los tres tipos de visualizaciones.

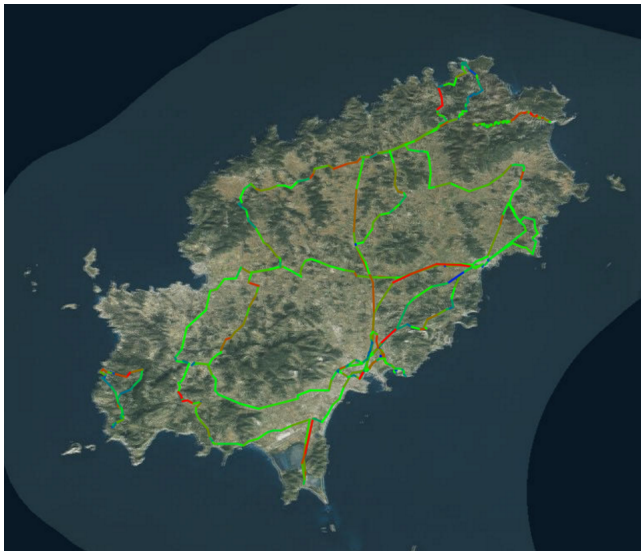


Figura 9.1: Trayectorias ciclistas de Ibiza sobre un mapa interactivo de OpenStreetMap

Este último capítulo se compone de cinco secciones. En la primera se da una introducción a la librería *matplotlib* para explicar qué es una figura y de qué partes se compone. En la segunda sección se detalla en profundidad cómo se lleva a cabo la gestión de los puntos y las trayectorias cuando se quiere crear una visualización, explicando la estructura de datos implementada para que la exportación se realice lo más rápido posible con la mayor cantidad de datos que se desee. Las secciones tres, cuatro y cinco explican cada una de las visualizaciones que se pueden crear en *Track Animation*: vídeos, imágenes y mapas interactivos, respectivamente.

9.1 Uso de la librería *matplotlib* en *Track Animation*

matplotlib es una librería que tiene como objetivo crear figuras y gráficas de visualización de datos de una forma rápida y sencilla. Esta librería es la base con la que se han implementado todas las visualizaciones por su facilidad de personalización e integración con otras librerías.

La base de *matplotlib* son las figuras, a partir de las cuales se crean los demás componentes que componen la visualización de los datos 9.2. Estos componentes son los *Axes*, títulos, leyendas, *canvas*, etc. Los *canvas* son útiles para sacar el dibujo final de toda la figura cuando esta es renderizada, pero para el usuario debe ser prácticamente invisible. Los *Axes* es una región de la figura que contiene los datos. Una figura puede contener varios *Axes* y estos están formados por dos ejes (***Axis***), que son los límites de los datos. Estos ejes son los encargados de crear los ***ticks***, es decir, las marcas sobre ellos y sus respectivas etiquetas.

La forma más sencilla de crear una nueva figura es la siguiente:

```
1 fig = plt.figure()  # An empty figure with no axes
2 fig, axarr = plt.subplots(2, 2)  # A figure with a 2x2 grid of Axes
```

Algoritmo 9.1: Creación básica de una figura en *matplotlib*

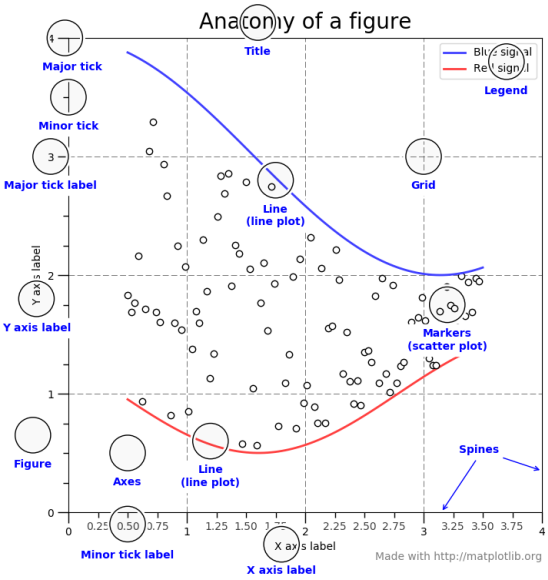


Figura 9.2: Partes de una figura de *matplotlib* [1]

En el siguiente ejemplo, la primera llamada a la función *plt.plot* es la encargada de crear la figura, que automáticamente genera sus *Axes* con las medidas necesarias. Las siguientes llamadas a esta función usan los *Axes* ya creados añadiendo únicamente nuevas líneas.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  x = np.linspace(0, 2, 100)
5
6  plt.plot(x, x, label='linear')
7  plt.plot(x, x**2, label='quadratic')
8  plt.plot(x, x**3, label='cubic')
9
10 plt.show()
```

Algoritmo 9.2: Ejemplo de uso de *matplotlib*

Al instanciar la clase *AnimationTrack* de *Track Animation*, se crea una nueva figura con un número de *Axes* en base al número de objetos *DFTrack* pasados por parámetro, es decir, se pueden pasar varios conjuntos de trayectorias que se visualizan de forma independiente dentro de la misma figura. Los límites de estos son los indicados por las coordenadas máximas y mínimas de los datos de cada *DFTrack*, usando la función *getBounds*, para que la visualización quede centrada sobre los puntos de GPS que contienen. También es posible especificar si se desea ver la visualización con un mapa de fondo de *OpenStreetMap*.

```
1  def __init__(self, df_points, bg_map=True):
```

```
    # Create Axes depending on the number of df_points
    self.fig, self.axarr = plt.subplots(len(df_points), 1)

    self.map = []
    self.track_df = DFTrack()
    # For each Axes, specify the Axis limits
    for i in range(len(df_points)):
        df = df_points[i].getTracks()
        df.df['Axes'] = i
        self.track_df = self.track_df.concat(df)

        trk_bounds = df.getBounds()
        min_lat = trk_bounds.min_latitude
        max_lat = trk_bounds.max_latitude
        min_lng = trk_bounds.min_longitude
        max_lng = trk_bounds.max_longitude
        if bg_map:
            self.map.append(smopy.Map((min_lat, min_lng, max_lat, max_lng)))
            self.axarr[i].imshow(self.map[i].img, aspect='auto')
        else:
            self.axarr[i].set_ylim([min_lat, max_lat])
            self.axarr[i].set_xlim([min_lng, max_lng])
```

Algoritmo 9.3: __init__ simplificado de la clase *AnimationTrack*

9.2 Gestión de los puntos y trayectorias para las visualizaciones

Se han creado dos funciones para gestionar la forma en la que cada punto y cada trayectoria es visualizada sobre una figura de *matplotlib*. La primera de ellas, **computePoints**, traza punto a punto todas las trayectorias siendo muy útil para cualquier visualización que requiera usar el frame rate como, por ejemplo, los vídeos o la impresión de una imagen en un segundo en concreto. La segunda función, **computeTracks**, se encarga de trazar las trayectorias completas, por lo que tiene que utilizar la función *plot* un número muy reducido de veces en comparación con la anterior función, siendo muy útil para exportar imágenes o mapas interactivos de las trayectorias enteras.

El trazamiento punto a punto en la función *computePoints* es necesario para que cada trayectoria se visualice de forma individual sobre una figura de *matplotlib* llamando a la función *plot* por cada punto nuevo. Con este planteamiento surgen dos problemas:

1. No es posible añadir un nuevo punto a una línea ya creada, por lo que se debe borrar dicha línea y recrearla con el nuevo punto.

2. No es posible pintar dos líneas diferentes con un solo *plot*, así que no se pueden trazar dos puntos de dos trayectorias diferentes aunque estos se hayan generado

al mismo tiempo o tengan que ir en el mismo fotograma.

Por este motivo se ha creado una estructura de datos que consiste en un diccionario de Python, *track_points*, que contiene por cada trayectoria los puntos ya trazados. Cada posición de este diccionario es una trayectoria que tiene otro diccionario, *position*, con únicamente dos elementos: la latitud y la longitud de cada punto. Estos dos elementos son listas de coordenadas que se le pasan a *matplotlib* para hacer un *plot* de los puntos ya trazados de una trayectoria en concreto [Figura 9.3]. A esta estructura de datos se le añaden trayectorias y puntos a medida que se itera el dataframe.

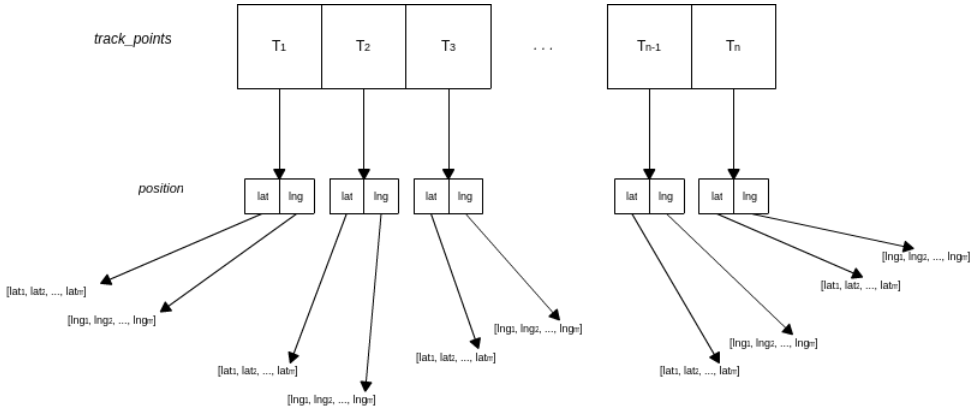


Figura 9.3: Estructura de datos implementada para la gestión del trazamiento punto a punto en las visualizaciones

La función *computePoints* se compone por un bucle principal que recorre cada uno de los puntos del dataframe y los va añadiendo a la estructura de datos para hacer el *plot*. Dado que el cómputo de este algoritmo es punto a punto, la función crea un generador que devuelve, en cada iteración, el punto trazado en ese momento y el siguiente en el dataframe. Esto se hace usando la palabra clave *yield* en lugar de *return*. Los generadores son iteradores que solo pueden ser usados una vez ya que no guardan todos los valores en memoria, sino que los genera en tiempo de ejecución. La primera vez que se llama al generador en un *for*, se ejecuta el código de la función desde el principio hasta que encuentra el *yield* en el bucle principal, devolviendo el punto que ha trazado en ese momento y el siguiente en el dataframe. En la siguiente iteración que se haga del generador, se ejecuta de nuevo el bucle de la función devolviendo los siguientes puntos.

```
1  def computePoints(self, linewidth=0.5):
2      track_points = {}
3
4      points = self.track_df.toDict()
5
6      # Main loop of the function
7      for point, next_point in zip_longest(points, points[1:], fillvalue=None):
8          track_code = point['CodeRoute'] + "_" + point['Axes']
9
10         # Check if the track is in the data structure
```

```
11          if track_code in track_points:
12              position = track_points[track_code]
13          else:
14              position = {}
15              position['lat'] = []
16              position['lng'] = []
17
18          lat = point['Latitude']
19          lng = point['Longitude']
20          if self.map:
21              lng, lat = self.map[int(point['Axes'])].to_pixels(lat, lng)
22
23          position['lat'].append(lat)
24          position['lng'].append(lng)
25          track_points[track_code] = position
26
27          self.axarr[int(point['Axes'])].plot(position['lng'], position['lat'], lw
                =linewidth)
28
29          yield point, next_point
```

Algoritmo 9.4: Algoritmo simplificado de gestión del trazamiento punto a punto

El algoritmo *computeTracks* es mucho más sencillo, ya que únicamente hace un *plot* por cada una de las trayectorias que hay en el dataframe. Esto es debido a que para representar una imagen final de las trayectorias no es necesario computarlas punto a punto, sino que basta con pasarle a la función *plot* la lista con sus coordenadas.

9.3 Generación de vídeos

La generación de vídeos con las trayectorias importadas es uno de los requerimientos esenciales por los que se ha creado la librería *Track Animation*. No obstante, la implementación de un algoritmo óptimo que compute una gran cantidad de puntos y genere los fotogramas necesarios para crear un vídeo ha sido una tarea compleja. Durante las primeras fases de su desarrollo se usaba el método *savefig* de la librería *matplotlib* para generar una imagen en *png* de cada fotograma y, posteriormente, unirlas mediantes FFmpeg. De esta forma, la creación de un vídeo de 691 puntos tardaba alrededor de 17 minutos debido a que la generación de cada *png* es una salida a disco duro que interrumpe la ejecución del algoritmo. Además, la función *savefig* realiza otras tareas y comprobaciones antes de guardar cada imagen. Otro de los motivos es que se generaba una imagen por cada uno de los puntos que había en el dataframe, aunque coincidieran en el tiempo. Por aquel entonces, aún no se había implementado la función de normalización [Capítulo 6].

La normalización ha permitido saber la cantidad de fotogramas que se crean y, además, que no haga falta generar una imagen por cada uno de los puntos del dataframe. Cuando el algoritmo encuentra una colección de puntos que deben ir en el mismo

fotograma, hace un *plot* de cada uno de ellos, como se ha explicado en el apartado anterior, pero genera una sola imagen. Además, la función está implementada de manera que guarda cada imagen en un *buffer* de memoria principal y pasa su información a una tubería (*pipe*) para que FFmpeg cree el vídeo al mismo tiempo que se ejecuta el algoritmo. Así que, normalizando los 691 puntos anteriormente comentados para que el vídeo dure 10 segundos a 10 FPS, salen 1028 puntos, creando el vídeo en 21 segundos.

Este apartado está destinado a explicar en dos subsecciones cuál ha sido la unión que se ha hecho de FFmpeg, el algoritmo *computePoints* y los *buffers* de memoria para hacer que la generación de los vídeos sea más rápida y eficiente.

9.3.1 Uso de la librería FFmpeg

FFmpeg es la librería que se usa para generar los vídeos a partir de las secuencias de imágenes creadas. Es de software libre y permite la manipulación, creación y conversión, entre otras operaciones, de audio y vídeo. En *Track Animation* se hace uso de FFmpeg en paralelo a la ejecución del algoritmo que gestiona el trazamiento de los puntos para una mayor velocidad, enviando las imágenes creadas a través de una tubería (*pipe*).

```
cmdstring = ('ffmpeg',
    '-y',
    '-loglevel', 'quiet',
    '-framerate', framerate,
    '-f', 'image2pipe',
    '-i', 'pipe:',
    '-r', '25',
    '-s','1280x960',
    '-pix_fmt','yuv420p',
    output_file + '.mp4'
)

pipe = subprocess.Popen(cmdstring, stdin=subprocess.PIPE)
```

Algoritmo 9.5: Creación de la tubería (*pipe*) para generar los vídeos con FFmpeg

Los argumentos que se usan de FFmpeg son los siguientes:

- **-y**: un archivo que ya exista con el mismo nombre es sobreescrito.
- **-loglevel**: especifica el nivel de información que se muestra por pantalla. El valor *quiet* no muestra ningún tipo de información.
- **-framerate**: indica la cantidad de imágenes de entrada que pasan por la tubería para generar un segundo del vídeo.
- **-f**: fuerza a que la entrada o la salida sea de una forma específica. En este caso, dado que está situado antes de la entrada (*-i*), fuerza a esta a que sea a través de la tubería.
- **-i**: indica que la entrada es a través de la tubería.

- **-r**: especifica la cantidad de fotogramas por segundo que tiene el vídeo final. La diferencia que tiene con *-framerate* es que, al estar indicado para la salida, duplica o borra fotogramas que hayan entrado dependiendo de si su valor es más alto o más bajo. Se ha usado para que todos los vídeos se vean con más calidad [Sección 6.2] a pesar del número de fotogramas que se especifique como entrada, siendo totalmente independiente de este ya que solo duplica o borra los fotogramas ya generados.
- **-s**: indica el tamaño que tiene el vídeo final.
- **-pix_fmt**: indica el formato de pixel. Por defecto, para la creación de vídeos *.mp4*, FFmpeg lo pone a *yuv444*, el cual no es compatible con la mayoría de reproductores que existen [46]. Por esta razón, se ha especificado *yuv420p*.

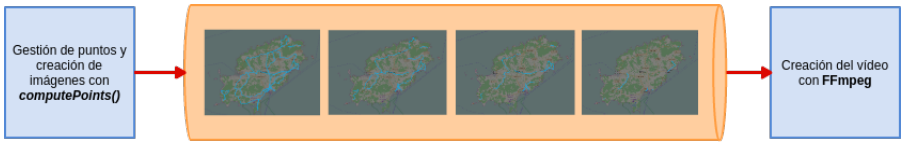


Figura 9.4: Tubería (*pipe*) por la que se envían las imágenes del *buffer* de memoria a FFmpeg

9.3.2 Uso de los *Buffers* de memoria en el proceso de creación de los fotogramas de un vídeo

Una vez creada la tubería para pasar las imágenes a FFmpeg, se crea un bucle con la función *computePoints*. Cabe recordar que esta función crea un generador que devuelve, en cada iteración, el punto trazado en ese momento y el siguiente en el dataframe. Gracias a esto y, haciendo uso de la columna *VideoFrame* del dataframe, es posible comprobar si el siguiente punto va en el mismo fotograma que el punto actual para generar únicamente los fotogramas necesarios y hacer que el vídeo dure el tiempo deseado en caso de que haya habido una normalización previa de las trayectorias.

La librería *matplotlib* tiene implementada la función *savefig*, la cual guarda la figura tal y como está en el momento de su llamada. Esta función realiza otras tareas y comprobaciones antes de guardar la figura, por lo que puede conllevar un tiempo adicional a la generación total del vídeo. Por este motivo, se ha implementado una forma de guardar en memoria principal las imágenes creadas haciendo uso del *canvas* de las figuras y el módulo *Image* de la librería *PIL* para convertirlas al formato PNG. Las imágenes en PNG creadas se guardan en un *buffer* de memoria que se pasa, posteriormente, a la tubería.

```
for point, next_point in self.computePoints(linewidth=linewidth):
    if self.isNewFrame(point, next_point):
        buffer = io.BytesIO()
        canvas = plt.get_current_fig_manager().canvas
        canvas.draw()
        pil_image = Image.frombytes('RGB', canvas.get_width_height(), canvas.
            tostring_rgb())
```

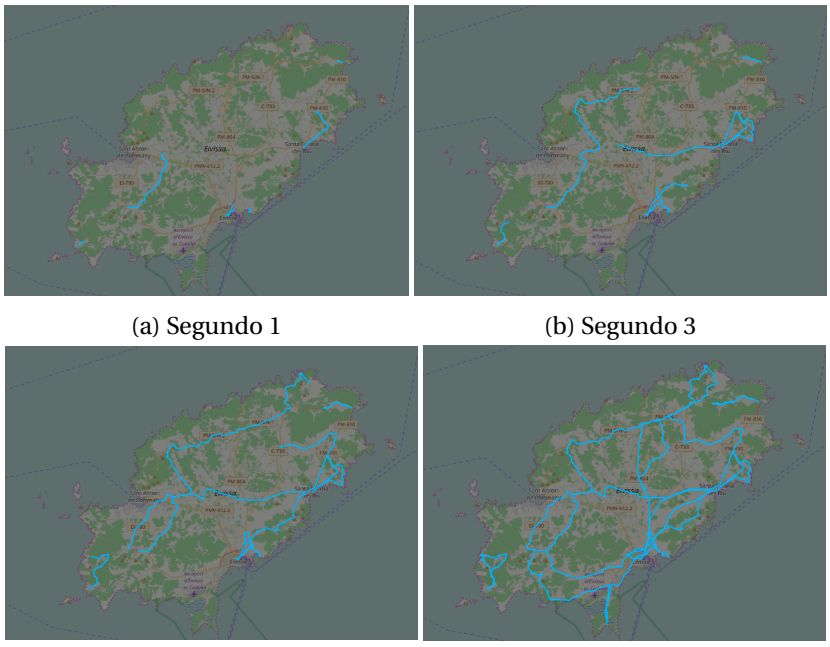
```
7         pil_image.save(buffer, 'PNG')
8         buffer.seek(0)
9         pipe.stdin.write(buffer.read())
```

Algoritmo 9.6: Creación de los fotogramas de un vídeo

Como se puede ver en el algoritmo, primero se comprueba mediante el punto actual y el siguiente, si se ha de crear un nuevo fotograma. En caso de que no, el algoritmo pasa al siguiente punto. En caso afirmativo, se crea el *buffer* de memoria en el que se guardan las imágenes generadas haciendo uso del módulo *io* de Python. Mediante la función *draw* de *canvas* de *matplotlib* se dibuja la figura para, posteriormente, convertir los píxeles en una imagen mediante la función *frombytes* de la librería *PIL*. Una vez generada la imagen, se guarda en el *buffer* en formato PNG, se apunta a la primera posición de este y se pasa la información a la tubería.

9.4 Generación de imágenes

El objetivo de *Track Animation* es el de crear diferentes tipos de visualizaciones de las trayectorias introducidas con el fin de poderlas interpretar de una forma más amigable y sencilla. Por este motivo, también es posible generar imágenes de las trayectorias, ya sea una imagen en la que todas ellas han finalizado como una o varias imágenes de segundos concretos que se especifiquen por parámetro. Cabe mencionar que, si se normaliza previamente, se conoce la duración que tiene un vídeo con un número de fotogramas en concreto y, por lo tanto, esto se puede aprovechar para sacar una imagen de cómo estarían las trayectorias en un determinado segundo.



(a) Segundo 1 (b) Segundo 3

(c) Segundo 5 (d) Segundo 10

Figura 9.5: Secuencias de imágenes por segundo

Para la generación de imágenes se ha creado la función *makeImage*. A esta función se le puede especificar mediante parámetro los FPS con los que se han normalizado las trayectorias y un segundo o una lista con segundos en los que se quiere crear una imagen. En el caso de no especificar ningún segundo, se entiende que lo único que el usuario desea es sacar la imagen final de todas las trayectorias, por lo que la función usada para tal fin es la de *computeTracks*, ya que no es necesario computar punto a punto. En caso de que sí se especifiquen segundos, se usa la función *computePoints* para calcular punto a punto qué segundo se está procesando hasta que coincida con alguno de los especificados, momento en el que se usa la función *savefig* de *matplotlib* para generar la imagen. En este caso sí se usa la función *savefig* debido a que no se ha de generar una secuencia de imágenes y, por lo tanto, el coste computacional de esta es imperceptible.

9.5 Generación de mapas interactivos

Otro tipo de visualizaciones que se pueden crear con *Track Animation* son mapas interactivos generados por la librería *mplleaflet*, la cual es capaz de sobreponer una figura de *matplotlib* sobre un mapa de OpenStreetMap. Con este tipo de visualizaciones es posible alejarse o acercarse a las trayectorias y moverse por el mapa 9.6.

Figura 9.6: Mapa interactivo con distintos niveles de zoom

La función implementada para este fin es *makeMap*, la cual usa únicamente *computeTracks* debido que no es necesario recorrer punto a punto. Finalmente, la función

save_html de *mplleaflet* crea un fichero HTML con el mapa interactivo y las trayectorias procesadas encima de él.

CAPÍTULO 10

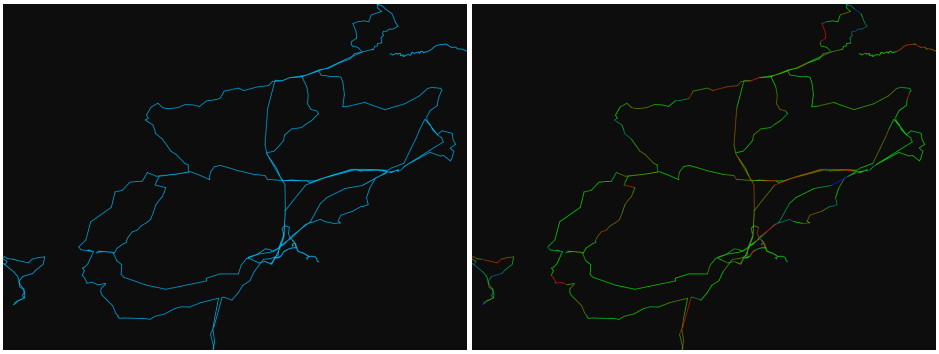
CONCLUSIÓN

En este documento he explicado cómo está hecha y cómo se utiliza la librería *Track Animation* de Python. El objetivo principal de *Track Animation* es generar visualizaciones de trayectorias de GPS almacenadas en ficheros con formato GPX de una forma en la que cualquier usuario final se quite de problemas y ataduras técnicas para usarla. Esto se ha conseguido gracias a la minuciosa implementación de una API con la que es posible manipular y procesar datos de trayectorias GPS para, posteriormente, exportar un vídeo, una imagen o un mapa interactivo. Sin ninguna librería que diera este tipo de soporte, todos estos procesos eran costosos.

Se ha proporcionado todo un conjunto de pautas para hacer uso de la librería *Track Animation*, explicando primero su arquitectura [Capítulo 3], seguido de sus funciones más básicas [Capítulo 5] y acabando con el conjunto de funcionalidades principales que permiten la manipulación y el procesamiento de los datos [Capítulos 6 - 9]. Estas últimas son la normalización de las trayectorias de GPS, los filtrados que se aplican a estas por fecha, tiempo y lugar, la aplicación de colores a cada uno de sus puntos a partir de los valores de un indicador y la exportación final de cada una de las visualizaciones. Se ha hecho uso de varias librerías de terceros para facilitar la implementación del conjunto de *Track Animation*, entre las que se destacan *gpxpy* [31] para la lectura y el parseamiento de los ficheros GPX, *pandas* [20] para la manipulación y el procesamiento de trayectorias usando dataframes, *matplotlib* [22] junto con *smopy* [37] para generar las visualizaciones de las trayectorias y añadir mapas de fondo y *geopy* [36] para obtener las coordenadas de lugares para poder filtrar las trayectorias.

Los ficheros GPX y CSV son los dos formatos de entrada de la librería *Track Animation*. Se ha observado que leer 1000 ficheros GPX de una media de 150 kB (más de un millón de puntos) tarda alrededor de 3 minutos, mientras que leer exactamente los mismos puntos guardados en un fichero CSV de 136 MB tarda 5 segundos. Esto se debe a que el CSV no tiene que parsear los tags que componen el XML. Los puntos leídos de los ficheros se pasan a un dataframe de la librería *pandas* para poderlos procesar antes de crear una visualización.

La normalización de las trayectorias es uno de los pilares más importantes que



(a) Resultado básico (b) Resultado con colores según velocidad

(c) Resultado básico con mapa de fondo

Figura 10.1: Ejemplo de diferentes visualizaciones que se pueden crear con *Track Animation*

componen el proyecto. Permite alargar o acortar las trayectorias con el fin de que todas empiecen y acaben al mismo tiempo sin importar su duración o distancia recorrida. Además, permite especificar el tiempo exacto que se desea que dure un vídeo con esas trayectorias importadas a *Track Animation*, añadiendo nuevos puntos en caso de ser necesario para dar un efecto de continuidad y uniformidad a estas al ser visualizadas. Esto hace que los fotogramas por segundo tengan una gran importancia sobre el resultado de esta función y, por consiguiente, el vídeo exportado, ya que permite acortar el tiempo que hay entre dos puntos consecutivos. Dicho de otra forma, si no se hubieran tenido en cuenta los FPS se verían las trayectorias dando saltos de diferentes tamaños dependiendo de la diferencia en la que los GPS capturaron cada uno de los puntos.

Otra funcionalidad de la librería *Track Animation* es el filtrado de las trayectorias en base a la fecha, el tiempo o el lugar, permitiendo así crear diferentes conjuntos de trayectorias con un menor número de puntos que hacen más fácil la interpretación final sobre una visualización [Figura 10.2]. Para ello se ha hecho uso de la librería *geopy*, para obtener las coordenadas del lugar por el que un usuario quiere filtrar las trayectorias, y las series temporales de *pandas* para realizar el filtro por fecha y tiempo.

El objetivo de *Track Animation* es crear visualizaciones de las trayectorias importadas y, para ello, es posible exportar vídeos, imágenes y mapas interactivos. Se hace uso de *matplotlib* para crear figuras con los puntos de las trayectorias que, posterior-

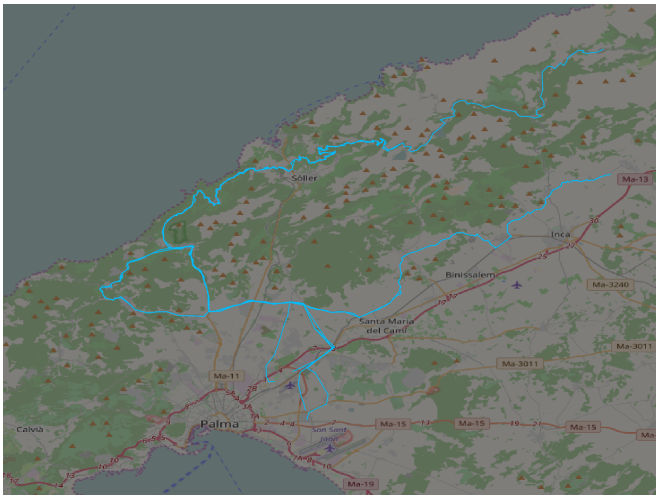


Figura 10.2: Trayectorias que pasan por Valldemossa entre las 08:00 y las 12:00 del primer cuatrimestre del año desde el 01/01/2010 hasta el 01/06/2014

mente, son guardadas en alguno de los 3 formatos comentados. Con *FFmpeg* [19] se generan los vídeos a partir de las imágenes que se crean y se almacenan en un *buffer* de memoria, haciéndoselas llegar a través de una tubería (*pipe*) para que sea totalmente independiente de la gestión de puntos que realiza el algoritmo implementado. Esto se debe a que guardar en disco cada imagen generada para cada fotograma del vídeo requiere un elevado tiempo de salida de información y una interrupción tras otra del algoritmo. Al hacerlo en paralelo, con los *buffer* de memoria y usando, además, la función de normalización, se puede realizar una exportación de un vídeo de 691 puntos en 21 segundos, mientras que guardar cada imagen en disco requiere de 17 minutos, aproximadamente.

10.1 Futuro trabajo

Debido al gran volumen de datos de GPS que se generan hoy en día y a la necesidad de aplicaciones para poderlos visualizar de una forma sencilla y rápida, *Track Animation* es un proyecto que tiene que continuar y, además, reproducirse a otras plataformas. Ya hay varias funcionalidades pensadas para implementar en próximas versiones como, por ejemplo, la lectura de otras fuentes de datos, la exportación de imágenes con metainformación geoposicionada que pueda ser importada de nuevo a *Track Animation*, la creación de mapas más personalizados y con diferentes estilos [47], la obtención del tiempo climático que había cuando se estaba haciendo la trayectoria [48] o la adición de etiquetas con información sobre la trayectoria en las visualizaciones.

Además, tal y como está implementada y estructurada la librería, es fácilmente reproducible a otro tipo de aplicaciones. Por ejemplo, se ha pensado en la implementación de un plugin para el programa QGIS [28] con el objetivo de que se pueda interactuar con las trayectorias usando al mismo tiempo otros plugins de terceros. Otro ejemplo es la creación de una página web con Django donde cualquier usuario pueda subir sus datos de GPS, interactúe y manipule las trayectorias mediante una interfaz

gráfica y, posteriormente, pueda ver los vídeos generados sobre la propia página web, descargarlos o compartirlos con otras personas.

Track Animation es una librería de software libre y cualquier persona puede aportar nuevas ideas e implementaciones con el fin de hacerla crecer de cada vez más.

10.2 Opinión personal

Track Animation es un proyecto del que me siento verdaderamente orgulloso por ser algo diferente a lo que los alumnos estamos acostumbrados a hacer en la carrera de ingeniería informática. El lenguaje de programación Python, la creación de una librería desde cero usando y uniendo otras, la manipulación de datos de GPS o la conversión de dichos datos a un formato visual animado, como pueden ser el vídeo o la imagen, es algo que nunca había hecho y, si no llega a ser por este proyecto, quizá no me hubiera ni planteado hacer. He disfrutado muchísimo durante todo el proyecto, siendo totalmente minucioso y perfeccionista en todo lo que hacía con tal de cumplir cada uno de los objetivos que me había planteado al inicio. Además, el haber aprendido a usar Python y la librería *pandas* ha provocado que empezara a trabajar en el departamento de *Big Data* de la empresa que estoy actualmente, lo cual es un gran objetivo en mi vida por ser algo a lo que sé que me quiero dedicar desde hace unos años.

Cada uno de los objetivos que se plantearon al inicio de este proyecto han sido cubiertos. *Track Animation* es un proyecto que puede seguir adelante y crecer, ya que no hay un número elevado de aplicaciones que dediquen tanto esfuerzo para que cualquier persona pueda usarlas de una manera tan sencilla y personalizable como lo hace esta. De esta manera, la esencia del proyecto es que la unión de varias librerías, como *pandas* o *matplotlib*, hacen que cualquier análisis que se esté llevando a cabo de trayectorias de GPS pueda ser importado a *Track Animation* para acabar generando una visualización.

No he hecho este proyecto pensando que iba a ser únicamente un trabajo de fin de grado, sino pensando en que es algo que puede tener futuro, que va a ser totalmente público y que cualquier persona va a poder utilizar. Es por esta razón que, a pesar de haberme divertido, ha habido momentos en que incluso rozaba la obsesión e invertía una gran cantidad de tiempo en seguir mejorando partes que ya estaban implementadas al mismo tiempo que creaba funcionalidades nuevas, y por el simple hecho de querer que quien use la librería lo haga a gusto y no piense en buscar otra que cumpla mejor sus requisitos. No obstante, estoy francamente contento de haber implementado una librería a la que, con pequeños cambios y en muy poco tiempo, se le pueden añadir nuevas funcionalidades, ya que hay una muy buena base creada.

BIBLIOGRAFÍA

[1] "Matplotlib usage." [Online]. Available: https://matplotlib.org/faq/usage_faq.html (document), 9.2

[2] W. McKinney, *Python for Data Analysis, 2nd Edition.* O'Reilly Media, 2017. (document), 7.2, 7.1

[3] J. Wang, Y. Mao, J. Li, and Z. X. andWen Xu Wang, "Predictability of road traffic and congestion in urban areas," 2015. [Online]. Available: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0121825 1

[4] S. Anthony, "Football: A deep dive into the tech and data behind the best players in the world," 2017. [Online]. Available: https://arstechnica.co.uk/science/2017/05/football-data-tech-best-players-in-the-world/ 1

[5] D. Orellana, A. K. Bregt, A. Ligtenberg, and M. Wachowicz, "Exploring visitor movement patterns in natural recreational areas," 2011. [Online]. Available: https://www.researchgate.net/publication/233752567_Exploring_visitor_movement_patterns_in_natural_recreational_areas 1

[6] Vespapp. [Online]. Available: http://vespapp.uib.es/ 1

[7] I. Lera, T. Pérez, C. Guerrero, V. M. Eguíluz, and C. Juiz, "Analysing human mobility patterns of hiking activities through complex network theory," 2017. [Online]. Available: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0177712 1

[8] A. Morrison, M. Bell, and M. Chalmers, "Visualisation of spectator activity at stadium events," 2009. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/5190776/ 1

[9] S. Khetarpaul, R. Chauhan, S. K. Gupta, V. Subramaniam, and U. Nambiar, "Mining gps data to determine interesting locations," 2015. [Online]. Available: https://www.researchgate.net/publication/254004719_Mining_GPS_data_to_determine_interesting_locations 1

[10] L. Ferrari and M. Mamei, "Identifying and understanding urban sport areas using nokia sports tracker," 2013. 1

[11] Garmin. [Online]. Available: http://www.garmin.com/es-ES/ 1, 2.3

[12] Tomtom. [Online]. Available: https://www.tomtom.com/es_es/ 1

[13] C. Coopmans and Y. Chen, "A general-purpose low-cost compact spatial-temporal data logger and its applications," 2008. 1

[14] D. Foster, "Gpx: The gps exchange format," 2016. [Online]. Available: http://www.topografix.com/gpx.asp 1, 1, 2.2, 4

[15] Bikely. [Online]. Available: http://www.bikely.com/ 1

[16] Wikiloc. [Online]. Available: https://es.wikiloc.com/ 1, 2.3

[17] Y. Zheng, L. Wang, and R. Zhang, "Geolife: Managing and understanding your past life over maps," 2008. 1

[18] Gpx viewer. [Online]. Available: https://play.google.com/store/apps/details?id=com.vecturagames.android.app.gpxviewer&hl=es 1, 2.3

[19] Ffmpeg. [Online]. Available: https://ffmpeg.org/ 7, 7, 9, 10

[20] Pandas. [Online]. Available: http://pandas.pydata.org/ 1, 5, 3.2, 4, 10

[21] Pandas time series. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/timeseries.html 1

[22] Matplotlib. [Online]. Available: https://matplotlib.org/ 1, 7, 3.2, 10

[23] E. Rampinini, G. Alberti, M. Fiorenza, M. Riggio, R. Sassi, T. O. Borges, and A. J. Coutts, "Accuracy of gps devices for measuring high-intensity running in field-based team sports," 2014. [Online]. Available: https://www.researchgate.net/publication/266153188_Accuracy_of_GPS_Devices_for_Measuring_High-intensity_Running_in_Field-based_Team_Sports 1

[24] K.-M. Cheung and C. Lee, "A trilateration scheme for relative positioning," 2016. 2

[25] A. Morales, "Los 10 formatos gis vectoriales más populares," 2017. [Online]. Available: https://mappinggis.com/2013/11/los-formatos-gis-vectoriales-mas-populares/ 2

[26] Gpx documentation. [Online]. Available: http://www.topografix.com/GPX/1/1/ 2.1

[27] Nasa world wind. [Online]. Available: https://worldwind.arc.nasa.gov/ 2.3

[28] Qgis. [Online]. Available: https://www.qgis.org/es/site/index.html 2.3, 10.1

[29] igo navigation. [Online]. Available: https://www.igonavigation.com/ 2.3

[30] Track road. [Online]. Available: http://www.trackroad.com/ 2.3

[31] gpxpy. [Online]. Available: https://github.com/tkrajina/gpxpy 2.3, 1, 3.2, 4, 10

[32] Gpx aviation. [Online]. Available: http://navaid.com/GPX/ 2.3

[33] Bbbike. [Online]. Available: http://www.bbbike.de/cgi-bin/bbbike.cgi 2.3

[34] Gps tracks. [Online]. Available: http://www.gps-tracks.com/ 2.3

[35] Trace gps. [Online]. Available: http://www.tracegps.com/ 2.3

[36] Geopy. [Online]. Available: https://github.com/geopy/geopy 5, 3.2, 10

[37] Smopy. [Online]. Available: https://github.com/rossant/smopy 7, 3.2, 10

[38] mplleaflet. [Online]. Available: https://github.com/jwass/mplleaflet 7, 3.2, 9

[39] tqdm. [Online]. Available: https://pypi.python.org/pypi/tqdm 3.1, 3.2

[40] Image. [Online]. Available: http://effbot.org/imagingbook/image.htm 3.2

[41] B. F. Janzen and R. J. Teather, "Is 60 fps better than 30? the impact of frame rate and latency on moving target selection," 2014. [Online]. Available: http://www.csit.carleton.ca/~rteather/pdfs/Frame_Rate_Latency.pdf 6.2

[42] L. M. Wilcox, R. S. Allison, J. Helliker, B. Dunk, and R. C. Anthony, "Evidence that viewers prefer higher frame-rate film," 2015. [Online]. Available: https://static1.squarespace.com/static/565e05cee4b01c87068e7984/t/57ae0ce42994ca2ad32bf74f/1471024357823/WilcoxAllisonetal+ACM+2015.pdf 6.2

[43] Q. L. andPeng Xu and H. Qu, "Fpsseer: Visual analysis of game frame rate data," 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7347633/ 6.2

[44] Bearing calculation. [Online]. Available: https://github.com/lforet/robomow/blob/master/navigation/gps/gps_functions.py#L223-L240 6.3

[45] Vincenty's formulae. [Online]. Available: https://en.wikipedia.org/wiki/Vincenty%27s_formulae 6.3

[46] H.264. [Online]. Available: https://trac.ffmpeg.org/wiki/Encode/H.264 9.3.1

[47] Folium. [Online]. Available: https://github.com/python-visualization/folium 10.1

[48] Wunderground. [Online]. Available: https://www.wunderground.com/weather/api 10.1