Universitat de les Illes Balears

escola politècnica superior

Final Degree Project

DEGREE OF COMPUTER ENGINEERING

Python library for tracing and animation of GPS tracks stored in GPX format files

JUAN JOSÉMARTÍNMIRALLES

Tutor

Dr. Isaac Lera Castro

Higher Polytechnic School

University of the Balearic Islands

Palma, June 2017

DEGREE OF COMPUTER ENGINEERING

Python library for tracing and animating stored GPS tracks

on files with GPX format

JUAN JOSÉMARTÍNMIRALLES

Tutor
Dr. Isaac Lera Castro

Higher Polytechnic School
University of the Balearic Islands
Palma, June 2017

My family and, above all, Espe for giving me all the love and
strength to achieve everything that I propose in my life.

To Edu, Paula and Sam, for listening to me speak for 4 months
only of this project and, even so, continue by my side.

To Carlos and Camila, for an incredible career
it would have been more difficult without you.

My grandfather for being, simply, the best grandfather
that could exist in this world.

Í GENERAL INDEX

Í INDEX OF FIGURES

Í INDEX OF TABLES

Í ALGORITHM INDEX

TO CHRONYMS

GPS	Global Positioning System
GPX	GPS eXchange Format
CSV	Comma-Separated Values
API	Application Programming Interface
CHALK	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

TO BSTRACT

Due to the great growth that the collection of all types of data has had in recent years, it is more necessary to have tools and programs that facilitate its analysis in a fast and efficient way. Both at the technological level and at the application level,
advances in data collection from **Global Positioning Systems (GPS)** It has been
made of getting bigger. On the one hand, there are mobile phones, bracelets and smart watches that have been improving the precision and the ability to capture GPS data. On the other hand, applications such as Google Maps have made the management and storage of this data easier for users.

Track Animation is the Python library created in this project to cover the need to analyze and visualize GPS tracks in a more friendly and simple way. Allows you to import GPS data stored in files with format **GPX**

and CSV to later analyze them based on the duration of the trajectories, elevation, speed or any indicator customized by the user through a color gradient. Furthermore, if what is desired is a more detailed visualization of the routes, there is the possibility of placing them on a **interactive map** in HTML format.

This document explains how the library has been implemented *Track Animation*
and how it works, detailing the different possibilities that exist to create an animation with all the desired trajectories and what a user should take into account to process the data in a fast and efficient way.

C CHAPTER 1

I NTRODUCTION

One of the great challenges that exists today is to be able to convert into knowledge all the large amount of geolocated data that is generated with Global Positioning Systems (GPS). Very diverse geopositioned information is stored, from daily vehicle traffic [3], movements of soccer players on the field [4], the preservation of natural spaces [5] or the detection of invasive species [6], the study of movement patterns [7], crowds at sporting events [8] or the points of interest that people visit the most [9]. Visualization is a very effective way to interpret the large volume of data generated in these and other related scenarios.

New technologies such as mobile phones and smart watches, among others, have made it really easy to record geolocated information, with sports being the one that has had the greatest popularity [10 , 7]. It should be said that previously there were already GPS devices, such as those of the Garmin brands [eleven] or Tomtom [12], but the low number of users who used them meant that the number of recorded trajectories was relatively low. With the cheapening of GPS due to the rise of smart phones, they have had to be reinvented and even today, they continue to survive with regularly updated free maps and better hardware. Technological progress is such that small GPS prototypes have emerged to carry out studies, such as SEAL [13], a device capable of capturing GPS data and storing it in GPX format [14] equipped with different CO2 or NH3 sensors.

At the same time, in the field of software, several websites such as Bikely [fifteen], Wikiloc [16] or Google Maps have made it possible for users to easily save, share and view all that information captured by GPS. For example, GeoLife [17] was a web application in which routes could be uploaded to be processed and displayed on a map with additional statistical information about them. Also, Android and IPhone mobile applications such as *GPX Viewer [* 18] are capable of reading GPS data sources from Google Maps, Mapbox, HERE, Thunderforest and OpenStreetMap, as well as files with gpx, kml, kmz and loc formats, and showing different indicators and graphs of these.

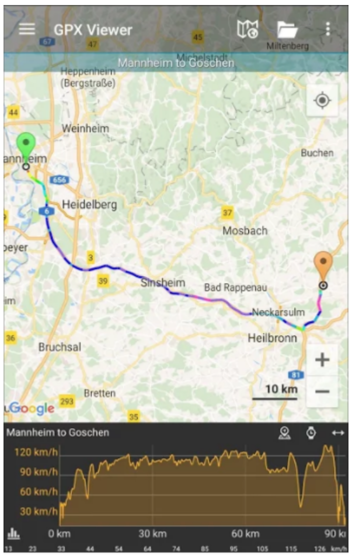


Figure 1.1: Application *GPX Viewer* for analysis and visualization of GPS data

Today, it is not uncommon to find various applications and data sources capable of saving, processing and displaying all this GPS information using different metrics such as altitude, speed, heart rate, power or air temperature, among others. .

Track Animation is the library created in this project and was born with the idea of carrying out more personalized and simple analyzes through visualizations such as images or videos. It is a library implemented in Python with which it is easily possible to import GPS data from files with GPX format [14] or CSV, process them and add custom metrics to create an easy-to-interpret interactive tour map, image or video. Any type of user, even with little programming knowledge, can create their own trajectory analysis, since the main objective of the implementation of this library is to give it a high level of usability and facilitate its use through an exhaustive design of its API to subtract end users from complex deployments and analyzes to visualize their trajectories.

The essential requirements it meets *Track Animation,* and that are explained in detail in their respective chapters of this document, are:

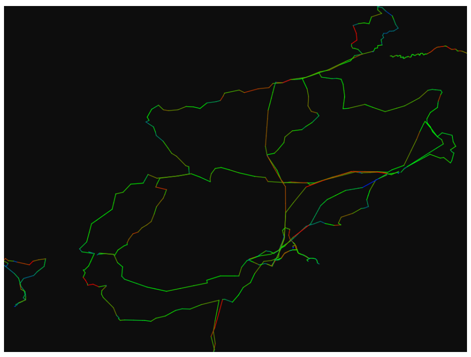
1. Import the trajectories of a file or a group of files in GPX or CSV format.
2. Join several sets of different trajectories in order to generate a single visualization with them.
3. Delete duplicate points of a trajectory from its code, latitude and longitude.
4. Normalize the trajectories so that they all start and end at the same time by specifying the duration of the video to be exported and the number of frames per second it must have.

5. Filter the trajectories by place, date or time.

6. Use a color palette to assign each of the points that make up a trajectory a color based on an indicator that the user wishes, either one of those calculated when importing the files, such as speed or altitude. , or anyone calculated by himself.

7. Export a visualization of the trajectories in video format (using *FFMPEG* [19]), in image format not with an interactive map of *OpenStreetMap* in HTML format.

(a) Basic result


(b) Result with colors according to speed

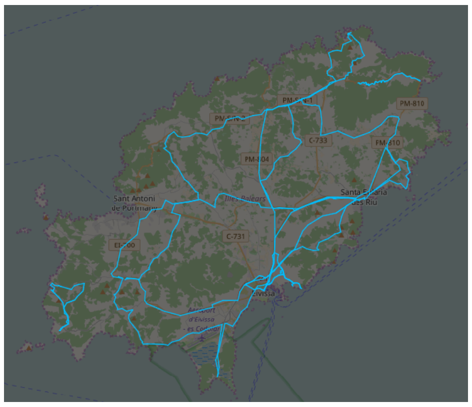

(c) Basic result with background map

Figure 1.2: Example of different visualizations that can be created with *Track Animation*

The bookstore *Track Animation* It is implemented in Python and, like its language, continues with the same philosophy integrating third-party libraries. One of these libraries is Pandas [twenty], which serves to facilitate data manipulation and analysis. It consists of two types of structures: series and dataframes. The latter are the basis of all the processing of trajectories of *Track Animation* since they allow, with great ease, to manipulate the data through numerical operations and time series [twenty-one]. Another of the libraries used is *matplotlib [* 22], and is generally used to create highly customized fi gures and graphs by the end user. The use that has been given in *Track Animation* is that of visualizing on a graph each one of the points of the processed trajectories using their geographical coordinates.

One of the major difficulties of the project has been the implementation of an algorithm for the optimal and efficient computation of the high number of points that make up the trajectories. This is because, in the first versions of the algorithm that were made, to form a video, an image was generated for each of the points to finally join them. This led to a lot of secondary memory outs, so its execution time was really long and inappropriate. By means of different techniques, which are explained throughout this document, it has been possible to reduce the execution time considerably, for example, from 17 minutes to 1 minute and a half for a data set of 691 points.

The precision with which the different GPS collects the data and the capture intervals for each one of them is another really important factor for the library and, therefore, has led to a preliminary study. One of the areas in which different studies have been carried out on this subject is in sports and, above all, in team sports, such as rugby or football, in which there are accelerations and decelerations with little time between them and where a very long interval between one GPS shot and another would cause a great loss of information. One of these studies is [2. 3], where they compare 2 GPS devices (SPI-Pro, GPS-5 Hz and MinimaxX, GPS-10 Hz) with a radar as a standard measure, using different metrics such as distance, speed and metabolic energy. The authors concluded that the accuracy of a GPS increases with higher sampling frequency but decreases as speed increases, in addition to that both GPS can be used to calculate distance or mean metabolic energy, but only the 10 Hz is capable of quantifying the distances covered at high speeds.

The project *Track Animation* it has been carried out in 6 phases. In the first, where the main objective of the library was already known, a search was made of the bibliography and related projects to be able to know what has been carried out on GPS in recent years and, more specifically, in layout and animation of trajectories. Later, in the second phase, an apprenticeship was carried out on the main libraries that make up *Track Animation: gpxpy, pandas* and *matplotlib.* In the third phase, small tests and implementations were made to find out the scope that could be achieved with each of the libraries and the computational costs of certain functions. The fourth phase was based on the compilation of the library's requirements *Track Animation* and the beginning of its implementation. In the fifth phase, at which time the implementation of the library was already consolidated, more functionalities were added that allowed greater usability and customization in the final displays. Finally, in the sixth phase the project documentation was written and certain functions were improved, both at the code readability level and at the computational level.

The objective of this document is to explain how the library has been implemented
Track Animation and what have been the research points, both at the level of GPS data and at the level of usability for any user, that have led to the library being implemented in the way it has been done. In addition, the best ways in which it can be used are analyzed and detailed to make it efficient and easy to use. This document consists of ten chapters of which, **from the fourth to the ninth, they explain**

each of the requirements that make up the library *Track Animation.* These
requirements are introduced in section 3.1, once the knowledge is explained

necessary to understand the project. These chapters are:

1. **Introduction:** shows and describes various projects related to *Track Animation,* Both at the hardware level and at the software level, the objectives for which this project has been created are detailed and a brief explanation of its requirements and its strengths and weaknesses are given.

2. **Global Positioning Systems:** because the project is based on ex-
To carry visualizations from GPS data, it is necessary to give a brief introduction of what a GPS is, what its characteristics are and in what types of files or databases its information can be stored. Furthermore, since the main source for importing data from *Track Animation* are the GPX files, it is explained in detail what they are and how they are composed.

3. **Bookstore architecture *Track Animation:*** one of the objectives of this project
The project is to give a high level of usability to the library *Track Animation* so that any user is exempt from technical difficulties to generate a visualization. For this reason, this chapter is focused on explaining in which packages, modules and classes the library is composed *Track Animation* and which third-party libraries it uses so that the algorithm explanations in the following chapters can be more easily understood.

Four. **Importing GPS data to *Track Animation:*** in this fourth chapter it is explained
the first part of the stream that GPS data passes through in *Track Animation:*
reading the GPX and CSV files.

5. **Basic library functionalities *Track Animation:*** the export of tra-
ectories to CSV and JSON, joining multiple data sets, and deleting duplicate points are the three basic functions of *Track Animation* explained in this chapter.

6. **Normalization of GPS tracks:** trajectory normalization is one of
the most important requirements of the library *Track Animation* since it allows a user to specify how long they want a video to have so that all the trajectories that compose it begin and end at the same time. This avoids the problems that coordinate sampling can cause in the visualizations. In addition, if necessary, new points are added to the trajectories in order to create an effect of continuity and uniformity in the videos when viewed.

7. **Filtered GPS tracks in *Track Animation:*** in order to give the user
Finally, greater usability and personalization of the visualizations, in *Track Animation* It is possible to filter the trajectories by place, date and time. This chapter is focused on explaining in detail how to make use of the functions implemented for this requirement and showing several examples of it.

8. **Applying gradient colors to paths:** another feature
of *Track Animation* is to draw the trajectories in the visualizations with color gradients according to the values of an indicator, either one from the import of data from the files or one of our own created by the user.

9. **Visualization of the trajectories processed in *Track Animation:*** the point f-
   end of the data flow passing through *Track Animation* is the visualization of the trajectories. As explained in this ninth chapter, these can be video, image or interactive map, based on two algorithms designed so that the final generation of visualizations is optimal and efficient to process large numbers of trajectories.

10. **Conclusion:** In this last chapter you can see the final conclusions of the project, milestones reached and future work for its continuation.

C CHAPTER 2

S ISTHEMS OF P OSITIONING G LOBAL

Global Positioning Systems (GPS) allow determining the position of an object on Earth from a few meters to 20 or 30 centimeters of precision if it is close to a reference station [24]. Trilateration is one of the mathematical methods used to determine the relative positions of objects using two or more reference points (visible satellites), and the measured distance between the object and each reference point. With three satellites it is possible to determine a two-dimensional position on the Earth through longitude and latitude, while with a fourth satellite it is possible to know the height at which an object is.

A *Geographic Information System (GIS)* It is a set of tools that allow the manipulation, analysis and modeling of any type of geographically referenced information. For this, the data must be stored in files or databases [25] among which are:

1. Shape fi le: is a set of at least three files (*shp, shx* and *dbf)* than a program ma GIS reads as one.
2. Spatial databases that can be *personal geodatabase* or from *Files of ESRI, PostgreSQL + PostGIS, Oracle Spatial* or *mySQL,* among other.
3. DWG - CAD format used primarily by AutoCAD.
4. GML: used to represent information on spatial elements from an XML format.
5. KML: also based on XML and developed for Google Earth, in 2008 it became the standard for the *OGC.* It is used to represent geographic data in three dimensions.
6. GPX: GPS Exchange Format can be used without the need to pay any license to define *points of interest, routes* or *trajectories* over an XML schema

also representing geographic data in three dimensions. Their great advantage is that they can be exchanged between different GPS devices.

The goal of the bookstore *Track Animation* is to create videos, images or maps of routes saved in a GPS data format. The GPX format is the one chosen for the main import of these to *Track Animation* for its ease of use and its flexibility to be exchanged between different GPS devices.

This chapter is intended to be a brief introduction to Global Positioning Systems (GPS) and to the GPX file format for storing your geopositioned data. It is made up of three sections. The first one gives a definition of *Waypoints, Tracks* and *Routes,* GPS data items that can be found within a GPX file. The second explains how GPX files are composed and how the information stored in them is structured. The third and final section gives a brief review of various libraries and applications that are compatible with GPX files today.

2.1 Points of interest, trajectories and routes of the files with GPX format

The GPX format (*GPS eXchange Format)* has been chosen to import GPS data to the library *Track Animation* Because it is free software, you do not pay any type of license, it is based on an XML format, which is very simple to parse, and it can be easily exchanged between different GPS devices. The first version 1.0 appeared in 2002 and 1.1, which is the one that has been used as a standard since then, appeared in 2004 [26].

GPX files can be made up of up to three types of objects: points of interest (Waypoints), trajectories (Tracks) and routes (Routes).

- ***WaypointType (* wpt):** It is a position within a collection of points that do not have a sequential relationship and that has the purpose of indicating a point of interest on a trajectory.

- ***TrackType (* trk):** is an ordered list of points that describe a path made up of at least one segment. This segment has a list of points (*TrackPoints)* they are connected. In case of loss of GPS connection, for example, a new segment is created within the path to continue with the path.

- ***RouteType (* rte):** it is an ordered list of points on a route. The difference with trajectories is that they do not store the locations of a journey that has been made, but rather the routes indicate a suggested path that has been specified in advance to a destination. They do not have dates or times but they do have estimates of duration or altitude.

For *Track Animation* only the trajectories have been taken into account (**trk),** that include the segments and the list of points that each one contains. However, for future versions, it is not ruled out that the other two types will also be added since the bookstore *gpxpy* it is able to read and parse them just like the trajectories. With this, it could be analyzed which are the most visited points of interest or make comparisons between the pre-established routes and the original routes.

2.2 Composition of GPX files

The GPX format is an XML schema composed of a series of common tags to store as much information as possible about a trajectory. The main tag of a GPX file is **gpx,** containing information such as name, description, author, etc. It must also include the GPX version (**version)** with which the file was created and the creator of this (**creator).**

The tag **gpx** it is the one that would contain the three types of data discussed in section 2.1, each with its own attributes. A path contains a list of segments (**trksec),** and these in turn a list of points (**trkpt).** A route is made up of a list of route points and a point of interest is only a point on the map. The possible position elements that each type of point can have are those specified in table 2.1.

Field	Units	Description
The t	Decimal degrees (WGS84)	Latitude
lon	Decimal degrees (WGS84)	Length
ele	Meters	Elevation in meters
time	Timestamp	Creation date and time in UTC format
magvar	Degrees	Magnetic variation
geoidheight	Meters	Height, in meters, of the geoid (sea level) above the ellipse of the earth.

Table 2.1: Position fields of a point (*TrackPoints)*

Latitude and longitude are the only required fields for a point, while the others are optional. In addition, it may contain other elements that are descriptive of the point, such as a name, description, or url, or precision information such as precision dilution to specify the additional effect on satellite navigation geometry in precision measurements. .

Dates and times are not local time, but use the system *Coordinated Universal Time (* UTC) with the format specified by ISO 8601, that is, it follows the criterion of specifying first the longest periods of time and then the shortest. For example, to specify a date we will first write the year, then the month, and finally the day [14].

An example of a file in GPX format would be the following:

```
< 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 >
                    < yam > 167 MALLORCA </ yam >
                    < cmt > Short route of the 312 Mallorca </ cmt >
                    < desc > Short route of the 312 Mallorca </ desc >
                    < trksec >
                              < 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 >
                              .
                              .
                              .
                    </ trksec >
          </ trk >
</ gpx >
```

Algorithm 2.1: Example of a file with GPX format

2.3 Libraries and applications compatible with the GPX format

Since GPX has become the standard for exchanging GPS data, many applications and libraries use it to read, share, analyze and visualize this data. Some applications or websites compatible with the GPX format have already been mentioned in the introduction to this document, such as, for example, *Wikilog [* 16] or *GPX Viewer*

[18]. However, some more examples are explained below with which you can see the different areas in which this format is used:

- **Google Earth,** that although it developed its own format (KML), it made the use of GPX compatible.

- **MapSource [** eleven] is a desktop application for transferring GPX files to Garmin devices.

- **Nasa World Wind [** 27] is a free software project to view satellite images of the world and to which different GPS file formats can be added.

- **QuantumGIS [** 28] is a program for creating, editing, visualizing and analyzing geospatial information.

- **iGo Navigation [** 29] is a GPS application for mobile devices to which routes and GPS tracks can be imported and exported.

- **Track Road [** 30] is a website with which routes can be planned and optimized in order to save fuel, time or money.

- **gpxpy [** 31] is a library for Python that reads and parses data from GPX files and with which you can easily get basic indicators such as the distance or speed between two points. It is the library used in *Track Animation* to import the GPS data.

In addition, from libraries or applications, you can find various data sources with free information and free of use such as *GPX Aviation [* 32], *BBBike [* 33], *GPS Tracks*
[3. 4] or *Trace GPS [* 35].

C CHAPTER 3

TO RQUITECTURE OF THE LIBRARY *Track Animation*

The bookstore *Track Animation* It has been created in order to be able to carry out a visual analysis of GPS trajectories from videos, images and interactive maps in a simple and comfortable way for any user, whether with or without programming knowledge. Giving a high level of usability to the library is one of the great objectives that have been taken into account when implementing it. Thanks to this, any user is exempt from technical difficulties to create a video with their trajectories

and, in addition, it has a wide range of possibilities to customize them. However, a certain knowledge of the API created in *Track Animation* and the structure that it has. For this reason, it has been decided to create this chapter in order to explain what the architecture of the entire library is like and what modules, classes and functions have been implemented.

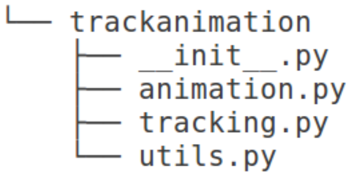
A Python library is simply a term that refers to a collection of source code that has been implemented for the purpose of being used by applications, scripts, other libraries, and so on. to offer specific functionality. Libraries can contain Python packages, which are nothing more than directories that, necessarily, must contain a file __ *init__.py,* which may or may not be empty. Packages are used to organize Python modules, which are files

. *py* that contain the source code. However, it is not strictly necessary for a module to be inside a package.

Track Animation is made up of a single Python package called *trackanimation.* This is composed of 3 modules where all the code of the library will be made up of 3 classes. The structure is as indicated in figure 3.1.

Modules *tracking* and *animation* They are made up of the following classes:

1. **ReadTrack,** inside the module *tracking.py,* used to read and parse a file or a set of files in GPX or CSV format to put them in a dataframe of the library *pandas.*



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

Figure 3.1: Structure of the modules that make up *Track Animation*

2. **DFTrack,** inside the module *tracking.py,* is responsible for treating and processing the data imported to the dataframe to normalize, filter or join them with more data from another source.

3. **AnimationTrack,** inside the module *animation.py,* is used to generate the videos, images or interactive maps, with the help of the library *matplotlib,* once the data has been read and processed.

The module *utils.py* has several concrete functions that are used by the library and a very simple class, called *TrackException,* to handle library exceptions.

With this structure it can be intuited that the library is composed of a class for data entry, another for data processing and another to generate an output, respectively.

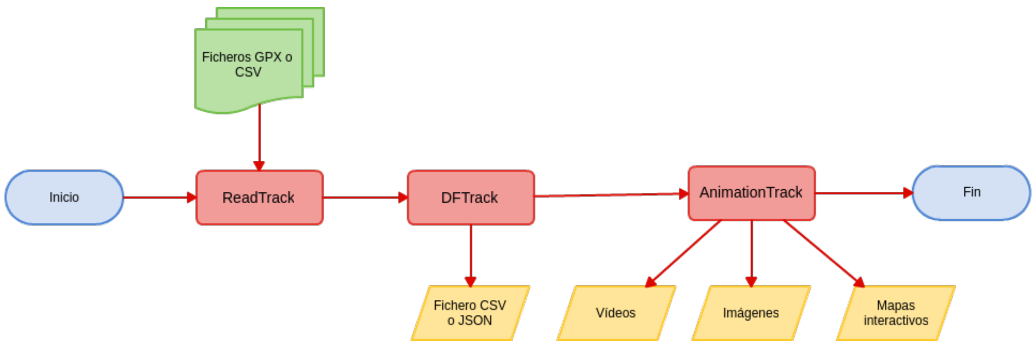


Figure 3.2: Diagram of the data flow in *Track Animation*

This chapter is focused on explaining what packages, modules and classes the library is made of. *Track Animation* and which third-party libraries it uses so that the explanations of the algorithms in the following chapters can be more easily understood. In the first section, the requirements of the library are mentioned again *Track Animation* so that, in the second section, it is easy to understand how third-party libraries have been used. The third, fourth and fifth sections are intended to explain each of the modules that make up the library *Track Animation* with their respective classes implemented.

3.1 Basic requirements of *Track Animation*

To understand why the library has been structured in this way, it is necessary to understand what can be done with it and what its requirements are. The final objective is that the trajectories previously imported from files in GPX format can be exported to videos, images or interactive maps. From the moment the trajectories are imported, until the visualizations are exported, the data is processed. This can consist of filtering the points by date, time or place, normalizing them or putting a color according to an indicator, be it speed, altitude or any other created by the user.

In summary, the basic requirements of the library *Track Animation* are as follows:

1. Import the trajectories of a file or a group of files in GPX or CSV format using the library *gpxpy [* 31].

2. Join several sets of different trajectories in order to generate a single visualization with them.

3. Delete duplicate points of a trajectory from its code, latitude and longitude.

4. Normalize the trajectories so that they all start and end at the same time by specifying the duration of the video to be exported and the number of frames per second it must have.

5. Filter the trajectories by place, date or time using the library *pandas [* twenty] and *geopy [* 36].

6. Give color gradients to each of the points that make up a trajectory from the indicator that a user wants, either one of those calculated when importing the files, such as speed or altitude, or any one calculated by him same.

7. Export a visualization of the trajectories in video format (using *FFMPEG* [19]), in image format not with an interactive map of *OpenStreetMap* in HTML format. This has been done using the library *matplotlib [* 22] in which the points are located on a graph by their coordinates, and *smopy [* 37] to put a map as a background image on the graph. The bookstore *mpllea fl et [* 38] has been used to pass the graphs of *matplotlib* to an interactive map.

In addition, the library has been used *tqdm [* 39] to see the progress of the algorithms implemented in the library when they are executed by command line.

3.2 Third party libraries used

To create *Track Animation,* Several third-party open source code libraries have been used in order to reuse code that is already created, taking advantage of the knowledge sharing that is made thanks to *open source.* Brushstrokes have already been given in previous chapters of the use that has been made of all of them, but in this section they are explained in more detail.

gpxpy: reading and parsing GPX files [31]

For reading and parsing the files in GPX format, the library has been used *gpxpy,* able to read these files and parse them creating different object lists of points of interest, routes, trajectories and the points that compose them. However, for *Track Animation* only trajectories have been taken into account.

pandas: processing of trajectories in dataframes [twenty]

pandas is a library to facilitate the manipulation and analysis of data. It consists of two data structures: *series* and *dataframes.* It is implemented on the library *NumPy.* Some of the many features it has is that it allows you to group data, filter it, convert it, join it, or use time series.

pandas is, next to *matplotlib,* the main library of *Track Animation.* Any trajectory of a GPX or CSV file read ends up in a dataframe with which its data is manipulated in the way the user wants to, finally, export the results, both in video, image, etc. CSV or JSON. The fact of having used this library in *Track Animation* It is because of the versatility in data management that it allows, being very useful to easily filter any point on any field, even making use of the time series on dates or times. It is for this reason that, thanks to the management of the time series that it has implemented, it has been possible to create two algorithms to filter the points by dates or by times. This is explained in more detail in section 7.2.

matplotlib: trajectory display [22]

matplotlib is a library that can be used to create custom graphs and figures such as histograms, bar diagrams, dot diagrams, etc. As has been said, *matplotlib* is the second largest library of *Track Animation,* since it allows to visualize each one of the points already processed in the dataframe of *pandas* on a graph. Starting from the latitudes and longitudes, join the points giving the possibility of personalizing the graph with different types of lines, various colors, with background images or with different sizes.

smopy: import maps to _matplotlib [_ 37]

To facilitate the interpretation of the exported videos and images, it is possible to add a static map of the places where the trajectories pass. For this, the library has been used *smopy* that allows, based on geographical limits, to download an image of a map of *OpenStreetMap* and use it together with *matplotlib*

by matching the points exactly where they pass.

mpllea fl et: **conversion of *matplotlib* interactive amapas [38]**

mpllea fl et is a library that gives the possibility of exporting figures from *matplotlib* to HTML by putting the trajectories on an interactive map of *OpenStreetMap.*

geopy: **Filtering trajectories by places [36]**

In *Track Animation* The possibility of filtering the trajectories by a specific place has been implemented just by indicating its name. For this we have used *geopy.*

geopy is a library that is implemented by different geolocation web services. It is very useful for locating the coordinates of addresses, cities or countries using third party geocoders and data sources. Some of the geocoders it uses, implemented in different classes, are OpenStreetMap Nominatim, ESRI ArcGIS, Google Geocoding API (V3), BaiduMaps, BingMaps API, OpenMapQuest or geocoder.us. This library can also calculate the distance between two points using the Vincenty formula or the Orthodromic formula with the package *geopy.distance.*

PIL: **conversion of *matplotlib* to images [40]**

PIL (Python Image Library) is a library that is used to interact with images, that is, to open, create, rotate or convert them to other formats. In *Track Animation* the module has been used *Image* from this library to convert a figure of *matplotlib* in an image in PNG format to later create a video.

tqdm: **progress bars on command console [39]**

tqdm is a library that creates progress bars on the command console in the loops of an application. Try to predict the time it will take to complete the loop and this will be more or less approximate as long as all iterations last the same and have a constant time. Has been added to *Track Animation* to give the user information about the progress of the processing, an approximation of what is left to finish and, above all, to give him more interaction.

3.3 Module *tracking.py*

The module *tracking.py* It is used to read and process trajectory data. It is composed of two classes: *ReadTrack,* which allows reading GPX and CSV files, and *DFTrack,* that allows to normalize, filter, concatenate or order trajectories.

3.3.1 Class *ReadTrack*

This class is in charge of reading from a CSV file or from a file or directory with several files in GPX format. Use the library *gpxpy,* which reads and parses the GPX files, and the library *pandas* to read the CSVs.

It is composed of 3 functions:

1. **readGPXFile:** is used to read a simple GPX file and store its points in a list.

2. **readGPX:** call the above function as many times as GPX files have the directory passed by parameter, or once in case it is a specific GPX file. Returns an instance of the class *DFTrack* with all the points read in a dataframe.

3. **readCSV:** is responsible for reading a single CSV file and returning an instance of the class *DFTrack.* It should be said that reading 1000 GPX files of an average of 150 kB (more than a million points) takes about 3 minutes, while reading exactly the same points stored in a 136 MB CSV file takes 5 seconds. This is because the CSV does not have to parse the tags that make up the XML.

3.3.2 Class *DFTrack*

The essential basis of the project is the class *DFTrack,* since it is the one that allows to customize the imported trajectories or the way we want the final visualizations to be. With it it is possible to perform many functions on the trajectories, such as filtering, normalizing, concatenating or exporting the trajectories to CSV.

This class receives as an input parameter a list of trajectory points that, through the library *pandas,* They are stored in a dataframe from which all the desired functionalities have been implemented in a very simple and organized way. Also use the library *geopy* to filter the trajectories by the places that are passed by parameter.

The bookstore *pandas* It also allows exporting the contents of a dataframe in CSV or JSON format, so a function has been created called **export** for this purpose.

The main methods implemented, and which are explained in different chapters of this document, are the following:

1. **getTracksByPlace:** passing it the name of a place by parameter, it returns all-
You give the paths that belong to it using the Google Geocoding API (V3) geocoders or, in case of error, OpenStreetMap Nominatim.

2. **getTracksByDate** and **getTracksByTime:** based on *Time Series* from the bookstore *bread-you give,* they are in charge of filtering all the points that belong to a date or a range of dates or times.

3. **timeVideoNormalize:** is one of the most important functions of the library and the one that allows that the creation of a video is more efficient, since it is possible to analyze more easily what is being seen. Its function is to normalize the points of all the imported trajectories in a dataframe so that they last in the video exactly the seconds indicated by the user, that is, that all the routes begin and end at the same time, although in reality it has not been the case. This is explained in more detail in Chapter 6.

Four. **setColors:** From an indicator of the trajectories, a color is assigned to each one of the points of the trajectories.

3.4 Module *animation.py*

The module *animation.py* it is in charge of taking the processed trajectories in video, image or interactive map format. It is made up of a single class called
AnimationTrack and the most important third-party library you use is *matplotlib*
to put each of the points that make up the trajectories on a graph. When instantiating this class, an instance of the class is passed as a parameter *DFTrack* with all the trajectories, optionally the number of points per inch that the visualization must have and a Boolean indicating if you want to add a static map of the places through which the trajectories pass as a background image.

The class consists of the following functions:

1. **computePoints:** is the function that plots point to point on a figure of *mat-plotlib* all trajectories, being very useful for any visualization that requires the use of the frame rate, such as, for example, videos or the printing of an image in a second determined by the user.

2. **computeTracks:** is responsible for plotting the complete trajectories, so it has a much lower computational time compared to the previous function, being very useful to export images or interactive maps of the entire trajectories.

3. **makeVideo:** create a video of the paths.

Four. **makeImage:** creates a final image with all the trajectories being able to specify Also, have images be created in one or more specific seconds in which you want to see how the trajectories are.

5. **makeMap:** creates an interactive map with the trajectories processed using the bookshop *mpllea fl et.*

3.5 Module *utils.py*

Since there are functions that are only used by library methods *TrackAnimation,*
This module has been created that contains basic and specific functions to perform a certain action. These are:

1. **getPointInTheMiddle:** calculates a new point between two dependent coordinates giving a specified distance and direction. It is useful to normalize the trajectories by time.

2. **rgb:** calculates a value from a minimum and a maximum to get a color in RGB. It is used to create the color gradient at the points according to an indicator.

3. **isTimeFormat:** checks if the passed value has a time or date format. cha. It is used in the functions to filter by ranges of times and dates to control the input that users give.

Also, this module has the class *TrackException,* which is used to throw the exceptions that may be in the library.

C CHAPTER 4

I DATA IMPORTATION GPS TO *Track Animation*

Importing GPS data to *Track Animation* It is an important part of the project because a good reading is necessary to be able to process the data correctly. The main GPS data storage format used in the library is GPX [14], which is a standard based on XML and easily interchangeable between different GPS devices, as already mentioned. To read this type of files and parse them, the Python library has been used *gpxpy [* 31], taking only the points of the trajectories and putting them in a dataframe of *pandas [* twenty].

Another format from which data can be imported is CSV. This has been done because the library *pandas* It allows you to export and import this format from dataframes and its reading is faster than GPX files, especially if the amount of data to be read is very large. For this reason, a good practice to use *Track Animation*

It would be to initially read the GPX files, process them and export them to CSV so that subsequent reads of the same data collection are made on that CSV and it is not necessary to process them again.

This chapter is made up of two sections whose purpose is to explain how GPX and CSV files are read in *Track Animation* and how the dataframes are initialized in the class *DFTrack* with imported data.

4.1 Reading files in *Track Animation* with *gpxpy*

To import GPS data from files with GPX format, the library has been used
gpxpy, which is capable of reading said files and parsing them creating different objects of points of interest, routes, trajectories and the points that compose them. For this, two functions have been implemented in the class *ReadTrack* of the module *tracking.py [* Section
3.3.1] with which to read a GPX file, a directory full of them or a CSV. It should be remembered that, although the bookstore *gpxpy* allows reading different types of points, only those of the trajectories have been taken into account.

Passing a GPX file to the library *gpxpy,* Since it is an XML, it first parses it and creates lists with the different objects mentioned. It is at this moment when the function is used *walk.* This returns a generator with all the points of the GPX file so that it is possible to go through it and save the information in a list to later convert it into a dataframe in the class *DFTrack.*

In *Track Animation* The data imported from GPX files can be classified into two types: those that are already written in the file, such as latitude, longitude, elevation or time, and those that have to be calculated at runtime, such as speed or time and distance traveled between one point and another. For the latter, 4 functions of *gpxpy: speed_between, time_difference, distance_3d*

and *distance_2d.* Of all these fields, there are some that are essential for *Track Animation* works properly and all its functions can be used [Table 4.1].

Name	GPX Field Units		Description
CodeRoute	-	String	GPX file name
Latitude	The t	Tenth grades them (WGS84)	Latitude
Length	lon	Tenth grades them (WGS84)	Length
Date	time	Timestamp	Creation date and time in UTC format

Table 4.1: Essential fields of a GPX file for *Track Animation*

Keep in mind that it is also possible to import data from CSV files, so it is necessary to have in the header the names of the fields as they are used in the library. *Track Animation,* that is, you have to use *CodeRoute, Latitude, Longitude,* etc. This is because the library itself is used to read a CSV *pandas,* which already converts the read data into a dataframe that is passed, later, by parameter to the class *DFTrack.*

4.2 Converting imported data to a dataframe of *pandas*

The bookstore *pandas* is the main library for data processing in *Track Animation.* All the points read from the trajectories are passed to a dataframe in said library from which different types of filters can be applied, the points normalized or several dataframes concatenated in a simple way.

All this processing is in the class *DFTrack* of the module *tracking.py.* There are three options to instantiate this class:

1. Do not pass any list or dataframe with dots, thus creating an empty dataframe.

Name	**GPX Field Units**		**Description**
CodeRoute	-	String	GPX file name
Latitude	The t	Tenth grades them (WGS84)	Latitude
Length	lon	Tenth grades them (WGS84)	Length
Altitude	ele	Meters	Elevation in meters
Date	time	Timestamp	Creation date and time in UTC format
Speed	-	km / h	Average speed between a point and next
TimeDifference	-	Seconds	Seconds difference between one point and the next
Distance	-	Meters	Distance traveled between one point and the next

Table 4.2: Fields read or calculated from a GPX file

2. That a dataframe is passed to it with the points of the inserted trajectories and with the names of the columns specified in tables 4.1 and 4.2.

3. Only pass a list of points, so the dataframe has the column names specified in table 4.2. There is also the option of passing a list with the names of the columns you want, taking into account that, at a minimum, there must be the fields of table 4.1.

The method __ *init*__ of the class *DFTrack* is the following:

```
def __init__ ( self , df_points = None, columns = None):
        if df_points is None:
                self . df = DataFrame ()

        if isinstance ( df_points, pd.DataFrame):
                self . df = df_points
        else :
                if columns is None:
                        columns = [ 'CodeRoute' , 'Latitude' , 'Longitude' , 'Altitude' , 'Date' ,
                                  'Speed' , 'TimeDifference' , 'Distance' , 'FileName' ]
                self . df = DataFrame (df_points, columns = columns)
```

Algorithm 4.1: Method __ *init*__ of the class *DFTrack*

C CHAPTER 5

F BASIC UNIONALITIES OF THE LIBRARY *Track Animation*

Though *Track Animation* is a library that aims to export animations of GPS trajectories, functionalities that complement it have been implemented to be able to make a wider use of it and have different ways to customize said animations.

This chapter is focused on explaining three basic functionalities of the library *Track Animation* to help understand some of the algorithms that are discussed in this document. These functions are:

- **export:** export of trajectories to CSV and JSON.
- **concat:** concatenation of sets of trajectories.
- **dropDuplicates:** deleting duplicate points in a toolpath.

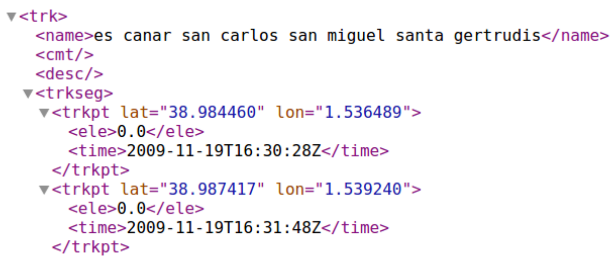
5.1 Exporting trajectories to CSV and JSON

A dataframe from the library *pandas* allows to be exported to different formats such as, for example, CSV, JSON. In *TrackAnimation* These functions have been used by joining them in a single method called *export* to which you only have to specify the format in which you want to export and the name of the resulting file. It is also possible not to pass any parameters to the function, which by default exports to CSV format under the name of *'exported_ fi le'.*

```
def export ( self , export_format = 'CSV' , filename = 'exported_file' ):
    if export_format == 'JSON' :
        self . df.reset_index (). to_json (orient = 'records' , path_or_buf = filename + '.
            json ' )
    elif export_format == 'CSV' :
            self . df.to_csv (path_or_buf = filename + '.csv' )
        else :
            raise TrackException ( 'Must specify a valid format to export' , "'% s'" %
                export_format)
```

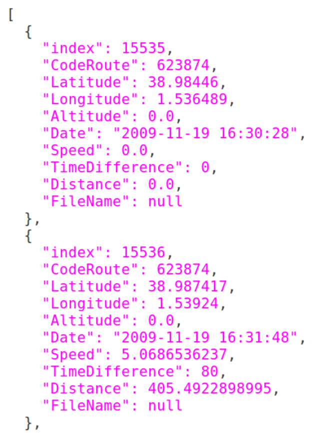
Algorithm 5.1: Exporting a dataframe from *TrackGPX*

The use of this function can be very useful if you want to create a set of trajectories filtering by date or place and then perform an analysis in an external application. Another utility is that if you have a large collection of GPX files and have to perform different executions for the same data collection, it can be very tedious to read all the files each time, so a good practice would be to perform a first execution where the desired data set is created, exported to CSV and that in each subsequent execution said CSV is read. As already mentioned, reading 1000 GPX files of an average of 150 kB (more than one million points) takes around 3 minutes, while reading exactly the same points saved in a 136 MB CSV file takes 5 seconds.



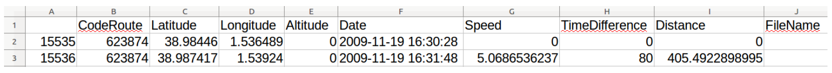
```
▼<trk>
   <name>es canar san carlos san miguel santa gertrudis</name>
   <cmt/>
   <desc/>
  ▼<trkseg>
    ▼<trkpt lat="38.984460" lon="1.536489">
       <ele>0.0</ele>
       <time>2009-11-19T16:30:28Z</time>
     </trkpt>
    ▼<trkpt lat="38.987417" lon="1.539240">
       <ele>0.0</ele>
       <time>2009-11-19T16:31:48Z</time>
     </trkpt>
```

(a) GPX format



```
[
  {
    "index": 15535,
    "CodeRoute": 623874,
    "Latitude": 38.98446,
    "Longitude": 1.536489,
    "Altitude": 0.0,
    "Date": "2009-11-19 16:30:28",
    "Speed": 0.0,
    "TimeDifference": 0,
    "Distance": 0.0,
    "FileName": null
  },
  {
    "index": 15536,
    "CodeRoute": 623874,
    "Latitude": 38.987417,
    "Longitude": 1.53924,
    "Altitude": 0.0,
    "Date": "2009-11-19 16:31:48",
    "Speed": 5.0686536237,
    "TimeDifference": 80,
    "Distance": 405.4922898995,
    "FileName": null
  },
```

(b) JSON format



	A	B	C	D	E	F	G	H	I	J
1		CodeRoute	Latitude	Longitude	Altitude	Date	Speed	TimeDifference	Distance	FileName
2	15535	623874	38.98446	1.536489	0	2009-11-19 16:30:28	0	0	0	
3	15536	623874	38.987417	1.53924	0	2009-11-19 16:31:48	5.0686536237	80	405.4922898995	

(c) CSV format

Figure 5.1: Example of GPX format with its equivalence in JSON and CSV format

5.2 Concatenation of sets of trajectories

In *Track Animation* it is possible to generate multiple instances of the class *DFTrack* in which each one of them has a dataframe with a different collection of data. Let's take as an example that from a collection where there are different cycling routes from all the Balearic Islands, we want to extract those that pass through Palma and Ibiza. Filtering must be applied twice per location: once for Palma and once for Ibiza. To generate a video in which both sets appear on the same map, it is necessary to join both dataframes

and, for this, the function *concat* to which it is necessary to pass as a parameter the object *DFTrack* that you want to join another object *DFTrack*. An example script to perform this task is as follows:

```
import trackanimation
desde trackanimation.animation import AnimationTrack

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

palm = bi.getTracksByPlace ( 'Palm' )
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' )

```

Algorithm 5.2: Script to concatenate two *DFTrack*

It is also possible to pass the function as a parameter *concat* a list with all *DFTrack* who wish to join another. In the following example Palma, Ibiza and Menorca are concatenated.

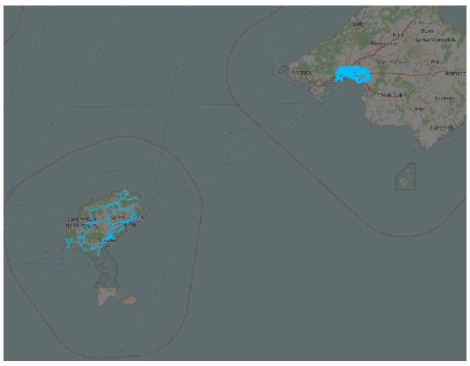
```
import trackanimation
desde trackanimation.animation import AnimationTrack

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

palm = bi.getTracksByPlace ( 'Palm' )
ibiza = bi.getTracksByPlace ( 'Ibiza' )
menorca = bi.getTracksByPlace ( 'Minorca' )

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' )

```

Algorithm 5.3: Script to concatenate a list of *DFTrack*

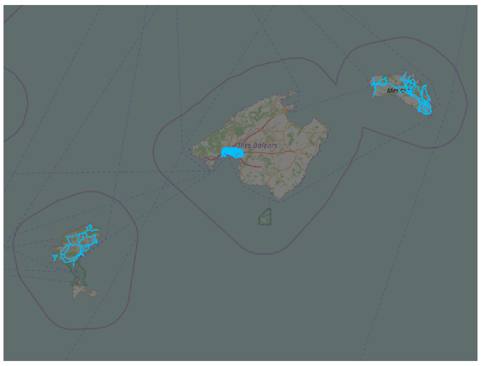
The results of both scripts can be seen in figure 5.2.

5.3 Deleting duplicate points in a toolpath

It is quite possible that a GPS is programmed so that each *x* seconds capture a point. If a user who is carrying out a trajectory stops for a few minutes and the GPS continues to collect information, they are repeated coordinates that in some cases cannot be useful and the only thing that causes the algorithm is slower. For this, the function *dropDuplicates,* which erases all the repeated points in the



(a) Points of Palma and Ibiza



(b) Points of Palma, Ibiza and Menorca

Figure 5.2: Trajectories concatenated using the function *concat* of *Track Animation*

dataframe from its name (*CodeRoute),* latitude *Latitude)* and length (*Longitude)* without needing to pass any parameters to it.

In another case that this function can be useful is when applying filters in places very close to each other, such as Palma and Marratxí. The coordinate limits of both places may have some deviation that can cause the same point to be in both sets.

C CHAPTER **6**

N ORMALIZATION OF TRAJECTORIES OF GPS

The main problem with saved GPS data is coordinate sampling, that is, the difference between one GPS device and another in the precision and capture interval of each point. This can lead to the final interpretations of one or more trajectories being unreliable and not comparable between them. In addition, the quality or poor con fi guration of some GPS to capture the points would cause the duration of a video exported with *Track Animation* it was too long or too short, depending on the number of points each trajectory had. It is for this reason that a point normalization function has been implemented in which it is possible to cause the displayed trajectories to start and end at the same time, regardless of their starting and ending times. Its main objective is that a user can control the duration of the videos and be able to study them more easily.

This function, called *timeVideoNormalize,* lengthens or shortens the trajectories based on the total time of each of them and the duration of the video specified by the user. This causes all of them to end and start in the video at the same time.

In addition, new points are added to give an effect of continuity and uniformity to the trajectories when viewing them in a video. To do this, it is calculated every how many seconds of the total duration of the trajectory some point must have, regardless of its total number of points. Thus, if the duration of a trajectory is 1200 seconds (20 minutes) and a user wants to obtain a video of 30 seconds, the trajectory must have at least one point every 40 seconds of this (1200s / 30s). In case there is no point within 40 seconds, a new point is created in proportion to the distance between the previous point and the next closest point. Therefore, the generated video has a minimum of 30 points in total distributed in at least 1 point for each second of this [See Figure 6.1]. In section 6.

The exact same algorithm could have been implemented using the total points of each trajectory. However, since the intervals between points for the same trajectory may be different, the visualization of these does not

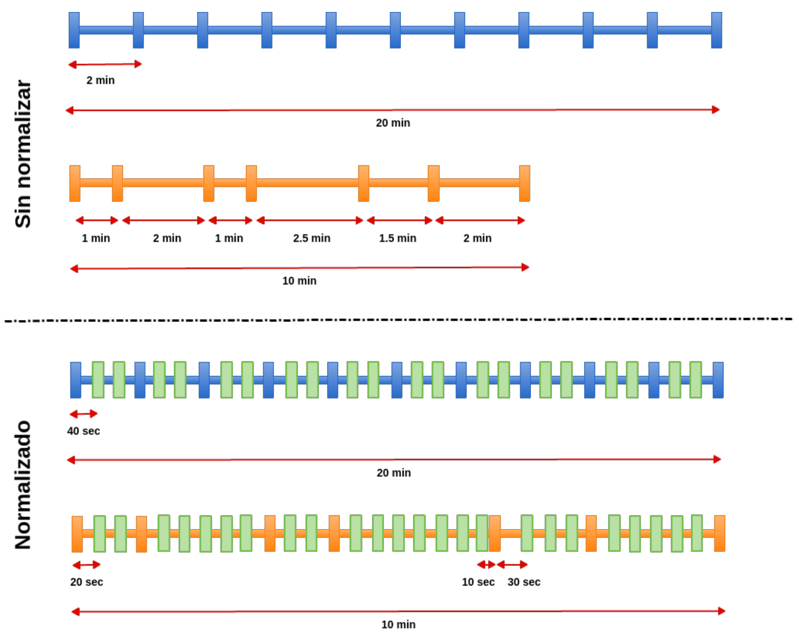


Figure 6.1: Comparison of two unnormalized and normalized trajectories

it is uniform and there are greater jumps the longer the interval and the fewer points there are in a trajectory. Also, there would be the same number of points in both a very small interval and a very large one. On the other hand, since the implemented algorithm only takes into account the total time of each trajectory, the points are uniform and practically the same distance between them since it ensures that each *x* seconds there is always some point.

This chapter is divided into three sections. The first one explains the basic concepts to understand what the normalization algorithm does. The second section shows the importance of frames per second in normalization to create an effect of uniformity and continuity to the trajectories. The last section explains how new points are created by performing some calculations to find out their coordinates.

6.1 Basic concepts of the normalization algorithm

One of the objectives of normalization based on the total time of each trajectory is to watch a video in which all trajectories, regardless of their duration or their total number of points, start and end at the same time. To do this, what has been done is to lengthen or shorten each trajectory based on the time in which you want the video to be seen.

The normalization algorithm goes through all the trajectories calculating, for each one of them, every how many seconds it must have at least one point and assigns the frame in which each of the points must go in the visualization in a new column of the dataframe, called *VideoFrame*. Furthermore, to solve the problem of the different intervals of each trajectory, the algorithm is able to insert as many new points between the previous and the next point in the interval as necessary, minimizing thus the length between these and making the trajectory more uniform. This is explained in depth in section 6.3. Two aspects must be taken into account for this function to have the desired effects:

1. The dataframe must have a field called *TimeDifference,* what is the difference in seconds between one point and another. This field is used to extract the accumulated time from the start of the trajectory to any point on it and use it as an "index" to be able to ignore the total points of the trajectory and base ourselves solely on its duration.

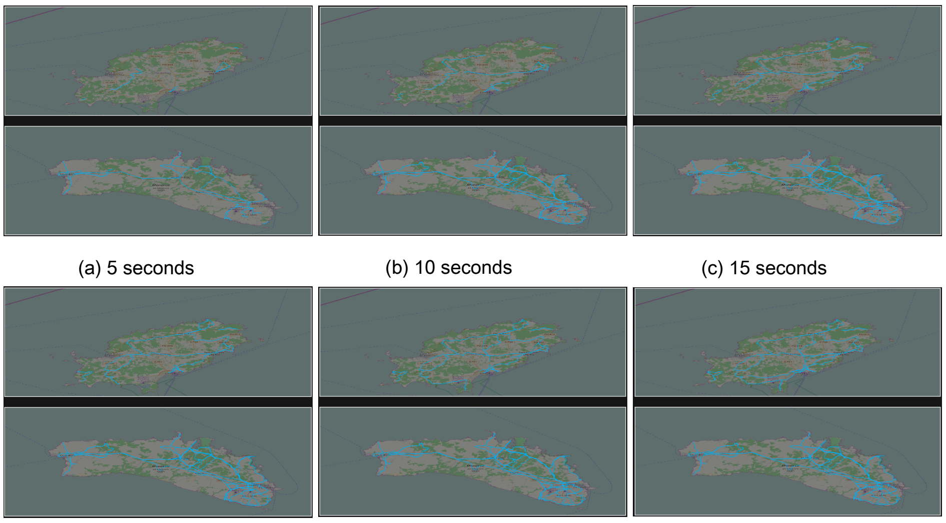
2. The normalization function must also be parameterized with the frames per second that you want the video to have. This causes the difference time in which there must be at least one point on the trajectory to be reduced, adding more intermediate points if necessary and reducing the intervals between them even more. In this way, more images are generated which, when joined to form the video, deceive the human eye by creating an effect of continuity in the trajectories. This is explained in more detail in section 6.2.

One of the functionalities that can be given to this method is that you can normalize a group of trajectories based on a desired time and then normalize another set of different trajectories as a function of another time. As a result, a video can be exported that compares both normalized sets, the duration of the video being equal to the time of the longest set. However, it must be taken into account that the frames per second for both normalizations must be the same, since otherwise the video would not have the desired duration. In the fi gure 6.2 you can see a comparison of two groups of trajectories where those of Menorca ended at 10 seconds, but the video continued until 30 seconds because it was specified that the trajectories of Ibiza should be normalized to last that time.

The same would happen if we concatenate both sets using the function *concat* of *Track Animation* and export the resulting dataframe. The difference would be that instead of seeing it in two different graphs, we would see it in one.

6.2 Importance of frames per second in normalization

In both the world of videogames and film, the images per second (*Frames Per Second (FPS)* or *frame rate)* have had a great relevance. FPS is the rate or speed at which the images (frames) of a video are played. Several studies have shown that the higher the FPS, the higher the playability, precision and quality of object movement [41 , 42 , 43]. For example in [41] it was demonstrated that a frame rate of 60 FPS gave 14% better results than a 30 FPS, but without major differences compared to a 45 FPS. Also in [42] it could be shown that the public greatly preferred a frame rate of 60 over one of 24 when watching movies. However, there were discrepancies between 48 FPS and 60 FPS between different types of playback and different movements.



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

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

Figure 6.2: Sequence of images from a video comparing cycling paths in Ibiza and Menorca

The frames per second have been taken into account when implementing *Track Animation,* giving the possibility to a user to choose with what frame rate he wants to export his video. This has been done due to the differences in which each point of the trajectories is captured. If the frame rate had not been taken into account, in every second of a video a jump would be seen between one point and another, and even more so the more capture interval there is between them. By means of the frame rate, what is caused is that the margin of seconds in which a trajectory should have at least one point is smaller, thus creating more new intermediate points. This gives a greater effect of continuity in the visualization of each trajectory.

Thus, continuing with the example of the introductory section of this chapter, if the duration of a trajectory is 1200 seconds and a user wants to obtain a 30-second video with a frame rate of 20, the calculation performed by the algorithm is

$$time\_diff = 1200\ s / 30\ s / \text{twenty}\ fps = 2$$

so the trajectory must have at least one point every 2 seconds of this. If not, a new point is created. In other words, the generated video will have 600 frames and therefore 600 points as a minimum. These are distributed in at least 20 points for each second of reproduction, that is, one point for each frame, which causes an effect of continuity in the trajectory in the human eye even if their capture intervals are very large or different. Keep in mind that if

time_diff is greater than the margin in which the GPS has captured the points, within the same frame there will be several of them. For this reason, the words "at least" or "at least" have been used when specifying the number of frames or dots per second that the video will have.

If you want to maintain the specified duration of the video, it is necessary that the same frame rate be indicated in the normalization and in the export of this. The same happens if you want to normalize more than one group of trajectories to put them in the same video.

6.3 Creating new waypoints

The normalization algorithm adds new intermediate points between two others when the intervals are longer than the time calculated in the normalization. Since each interval can be different from each other, adding a new point does not take into account the length and duration of the entire trajectory, but rather that of each of its intervals. In this way, the new point is proportional to the time and length of the interval and, therefore, to the trajectory itself. To do this, the proportional time in which the new point must go is calculated using the formula:

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

where *time_diff* is the time difference in which the trajectory should have at least one point, *point_idx* is the total number of new points that have already been created in that interval and *end_point ['TimeDifference']* indicates the total number of seconds in the interval.

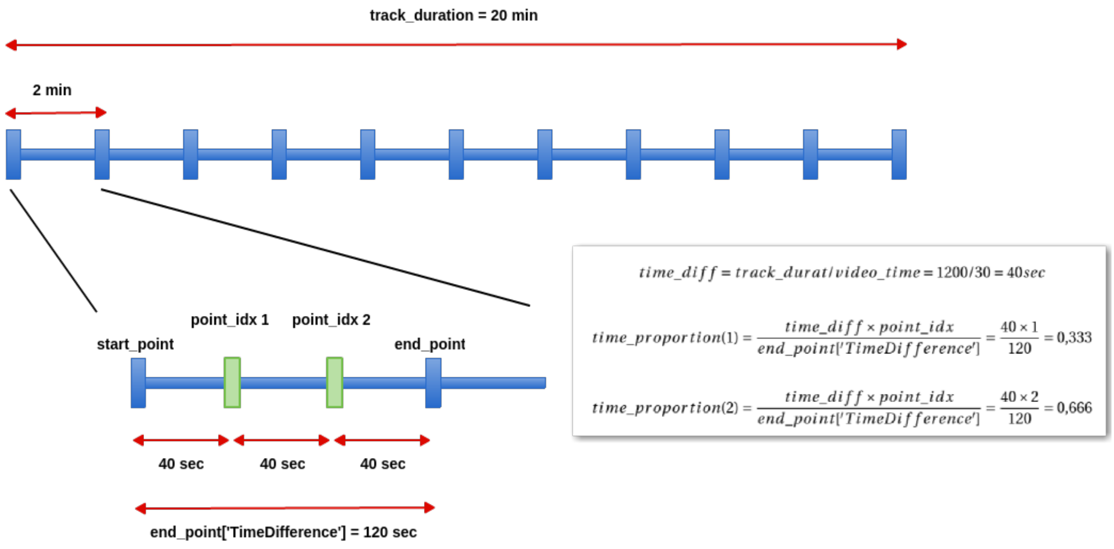

track_duration = 20 min
2 min
point_idx 1
point_idx 2
start_point
end_point
40 sec
40 sec
40 sec
end_point['TimeDifference'] = 120 sec
time_diff = track_durat / video_time = 1200/30 = 40sec
time_proportion(1) = time_diff × point_idx / end_point['TimeDifference'] = 40 × 1 / 120 = 0,333
time_proportion(2) = time_diff × point_idx / end_point['TimeDifference'] = 40 × 2 / 120 = 0,666

Figure 6.3: Adding new points to a path

Subsequently, the distance and the proportional time in which the new point in the interval must go is calculated. With this, it is already possible to calculate the relative speed at which the subject was going at that moment while doing the trajectory, the time and the relative altitude. Finally, using an object *Point* from the bookstore *geopy* With the coordinates of the initial point of the interval and another with those of the end point, the coordinates of the new point are calculated by specifying how far it is from the initial point. The simplified algorithm is as follows:

```
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
```

Algorithm 6.1: Creating a new waypoint

To calculate the coordinates of a new point, it is necessary to know its distance from the initial point and the direction in degrees in which the trajectory is going. For this, the algorithm of [44]. It should be said that, although the class is instantiated
VincentyDistance of *geopy,* Vincenty's formula has not been used [Four. Five] to calculate the distance between the initial point and the new one since, as seen in the previous algorithm, this is calculated by means of the proportional distance of the first point of the interval and the last one. Only the class is used *VincentyDistance* to pass the distance already calculated as a parameter. Subsequently, an algorithm from the library *geopy* calculates the new coordinates.

```
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)

```

```
    tan_start = math.tan (start_lat / 2.0 + math.pi / 4.0) tan_end = math.tan (end_lat / 2.0 + math.pi / 4.0)
    dPhi = math.log (tan_end / tan_start)

    bearing = (math.degrees (math.atan2 (d_lng, dPhi)) + 360.0)% 360.0;

    # ***********************************************
    # Get coordinates
    # ***********************************************
    distance_km = distance_meters / 1000
    d = geo_dist.VincentyDistance (kilometers = distance_km)
    destination = d.destination (point = start_point, bearing = bearing)

    return geopy.Point (destination.latitude, destination.longitude)
```

Algorithm 6.2: Calculation of the coordinates for a new intermediate point

C CHAPTER 7

F ILLUSTRATED TRACKS OF GPS IN *Track Animation*

Trying to interpret the evolution of the motion of a very large collection of trajectories can be a complex task. That is why one of the essential characteristics for conducting an analysis in any area is the creation of small subsets of data from a larger set. To obtain these subsets it is necessary to filter the data and process them until obtaining the desired information. A GPS track is made up of coordinate points that have a date associated with them, so being able to apply filters to these values is essential.

In *Track Animation* Three functions have been created with which to filter the trajectories by place, date and time. These criteria allow the analysis and comparison of more specific data, being able to apply different indicators and normalizations for their final display. These functions are *getTracksByPlace, getTracksByDate* and *getTracksByTime,*
respectively. The first one uses the geocoders of *GoogleV3* and *OpenStreetMap* from the bookstore *geopy* to return the points that are in a specific place or those of all the paths that have ever passed through that place. Filtering by date and time uses the tools for time series of the library *pandas.*

With these three functions it is possible, for example, to obtain a set of data with the points of Palma that exist between 09:00 and 12:00 of the first four-month period of the year from 01/01/2015 to 01/06 / 2017. The result of applying these filters can be seen in figure 7.1.

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

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

valldemossa = bi.getTracksByPlace ( 'Valldemossa' , only_points = False) 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' )

```

Algorithm 7.1: Filtering script of trajectories by place, date and time

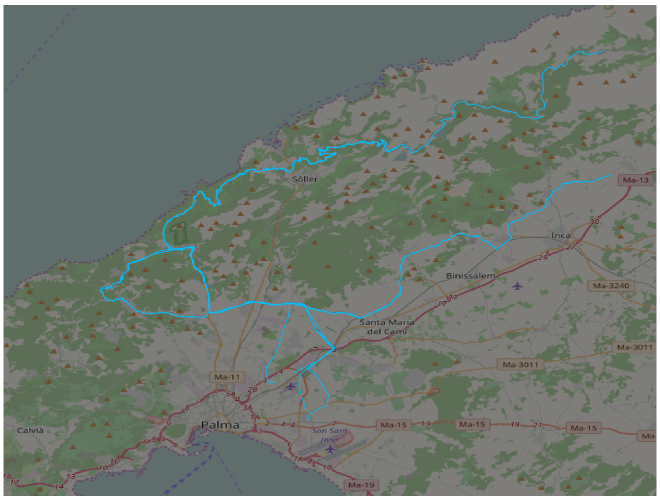


Figure 7.1: Trajectories that pass through Valldemossa between 08:00 and 12:00 first semester of the year from 01/01/2010 to 06/01/2014

This chapter is made up of two sections detailing how filtering by place and filtering by date and time has been implemented and how it works, respectively.

7.1 Filtering by place

The large number of trajectories shown in a video makes it difficult to correctly interpret the data. That's why analyzing smaller, place-focused data sets is essential. This allows multiple locations or different indicators for the same location to be compared as well. The created function is called *getTracksByPlace* and is in the class *DFTrack.* This function calls two other functions, *getTracksByPlaceGoogle* and *getTracksByPlaceOSM,* which are in charge of parsing the JSON that the APIs return to extract the limits of a place through Google services or Open Street Map services, respectively, using the library's geocoders *geopy.* The algorithm works is to call first to the Google API and in case it fails, give a *timeout* or does not return information, call the Open Street Map.

To the three functions it is possible to enter the following three parameters:

- **place:** the place from which you want to recover the points.
- **timeout:** the time the geocoder service waits to respond before returning a null value.
- **only_points:** Boolean in which you specify whether what you want to retrieve are only the points that pass through a place or the complete paths that have passed through that place.

Parameter *only_poins* it's really important because of the effect it can have on a visualization. If a set is created specifying that only the points that pass through a certain place are desired and then concatenated with another set from another place, it is possible that there are points of the same trajectory in both sets, which means that who made that trajectory went through both places. In the fi nal display, this can result in completely straight lines between one place and another because the intermediate points have been missed in the filtering. Let's take the case that we want to obtain the points that pass through Sóller and those that pass through Manacor. The final result is the one that can be seen in Figure 7.2.

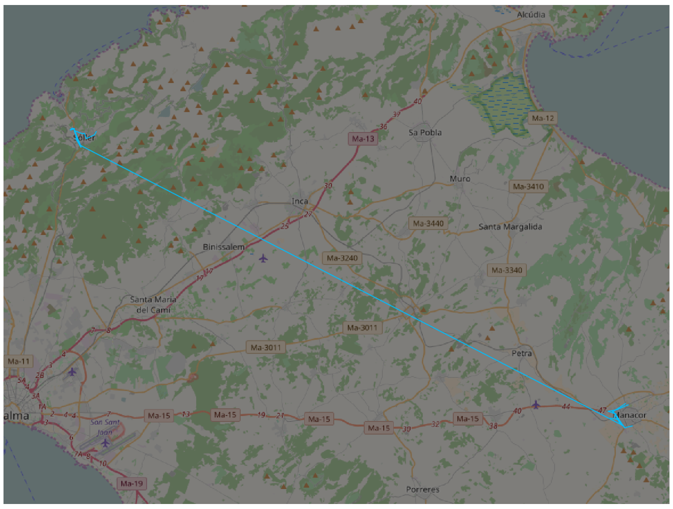


Figure 7.2: Points of Sóller and Manacor

In the case where the parameter *only_points* is set to false, in the fi nal display you will be able to see all the paths that have passed through a certain place [Figure 7.3].

From this function, it is possible to process the data in different ways in order to obtain other results. As already mentioned, the main object of the class
DFTrack is a dataframe containing trajectory points. This dataframe can be used outside the library itself *Track Animation* using the library's features and tools *pandas.* An example is being able to obtain only the trajectories that have passed through several places, as shown in the following script,

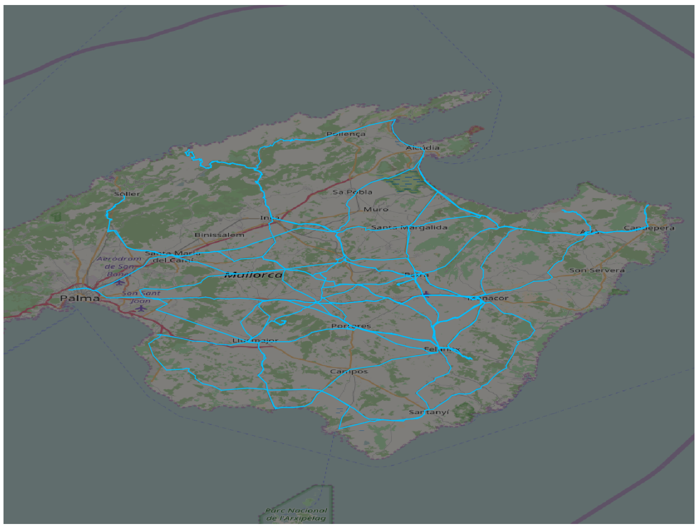


Figure 7.3: Trajectories that have passed through Manacor

where the final result [Figure 7.4] shows the points of Sóller and Manacor plus the trajectories that have passed through both places:

```
import trackanimation
desde trackanimation.tracking import DFTrack
desde 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' )
```

Algorithm 7.2: Script to visualize the points of Sóller and Manacor plus the trajectories that have passed through both places

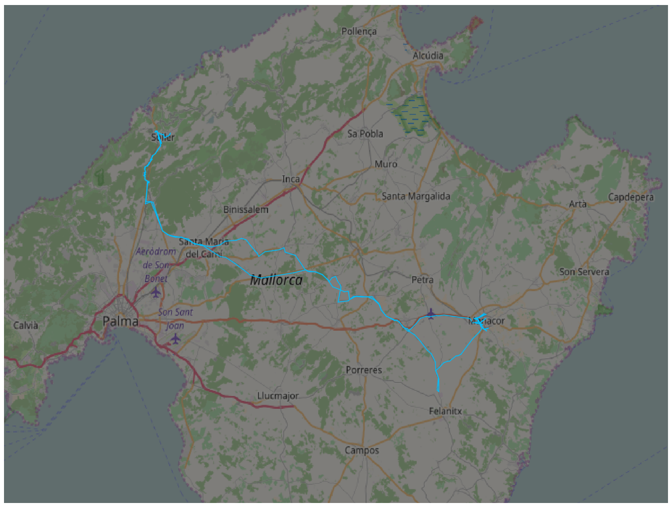


Figure 7.4: Points of Sóller and Manacor plus the trajectories that have passed through both places

7.2 Filtering by date and time

A simple way of filtering by date or time has been implemented to further narrow the data and analyze the trajectories based on time periods. For this, the tools for time series of the library are used *pandas,* which is nothing more than a structured form of temporal data that indicates a specific instant in time (timestamp), fixed periods such as a whole month or year, or time intervals from an initial timestamp to an end [2].

The time series of *pandas* They are used so that, from an already created data set, the date range established as an index of the set is added to partition, add, group, etc. the data from that index. However, what you want to do in *Track Animation* is that from a set of trajectories, those that belong to a period or interval of dates or times are selected. In other words, the time series tools of *pandas* have been used in *Track Animation* only for creating date ranges and not for creating time series data sets. In this way we ensure that the structure of the data frame does not change and, furthermore, it is possible that in the same implemented function the trajectories can be filtered by any of the three time groups mentioned in the previous paragraph.

For this, two functions have been created: *getTracksByDate* to filter by date interval or periods and *getTracksByTime* to do it from a range of times.

7.2.1 Function *getTracksByDate*

This function is in charge of filtering the trajectories by a range or an interval of dates from four parameters:

- **start:** start date of the period.
- **end:** end date of the period.
- **periods:** number of periods from the start date or until the end date.
- **freq:** frequency with which a new period is taken.

Only two of the three parameters can be specified *start, end* or *periods.* Parameter *freq* It has by default the value 'D', which means that the periods are daily. There are other values that can be used in this parameter and that are specified in table 7.1.

Alias	Type	Description
D	Day	Daily calendar
B	BusinessDay	Daily work calendar
M	MonthEnd	Last day of the calendar month
BM	BusinessMonthEnd	Last day of the month of the business calendar
MS	MonthBegin	First day of the calendar month
BMS	BusinessMonthBegin	First day of the month of the business calendar
W-MON, W-TUE ...	Week	Weekly dates on a specific day of the week
WOM-1MON, WOM-2MON ...	WeekOfMonth	Generate dates weekly in the first, second, third or fourth week of the month
Q-JAN, Q-FEB ...	QuarterEnd	Quarterly dates of the last calendar day of each month, ending in the specified month
BQ-JAN, BQ-FEB ...	BusinessQuarterEnd	Quarterly dates of the last day of the work calendar of each month, ending in the specified month

QS-JAN, QS-FEB ...	QuarterBegin	Quarterly dates of the first day of the calendar of each month, ending in the specified month
BQS-JAN, BQS-FEB ...	BusinessQuarterBegin	Quarterly dates of the first day of the work calendar of each month, ending in the specified month
A-JAN, A-FEB ...	YearEnd	Annual dates of the last calendar day of the specified month
BA-JAN, BA-FEB ...	BusinessYearEnd	Annual dates of the last day of the work calendar of the specified month
AS-JAN, AS-FEB ...	YearBegin	Annual dates of the first calendar day of the specified month
BAS-JAN, BAS-FEB ...	BusinessYearBegin	Annual dates of the first day of the work calendar of the specified month

Table 7.1: Frequencies admitted by the time series of *pandas [* 2]

The function *getTracksByDate* use the library method *pandas date_range* to which exactly the same parameters are passed. This function returns only specific dates (with format *YYYY-MM-DD)* according to specified dates, periods or frequencies. Subsequently, the date values of the GPS track points, which have a UTC date format (*YYYY-MM-DDThh: mm: ssz),* are converted to format *YYYY-MM-DD* to make them match the dates previously returned by the function *date_range.* The algorithm is as follows:

```
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 ( ' %AND -% m -% d ' )) df = df [df [ 'ShortDate' ]. apply ( lambda date: date in rng)]

        of the df [ 'ShortDate' ]

        return self .__ class __ (df, list ( df))
```

Algorithm 7.3: Function *getTracksByDate* of *Track Animation*

Some examples of using this function are the following:

```
"" "
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 - eleven - 30 ',
                              . . .
                              '2011 - 08 - 31 ',' 2011 - eleven - 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 = 'TO -JAN ' )
```

Algorithm 7.4: Examples of use of the function *getTracksByDate*

7.2.2 Function *getTracksByTime*

This function has been implemented so that, apart from filtering by dates, it can also be filtered by time intervals. For this we have used the function *indexer_between_time*
from the bookstore *pandas* to which an initial hour and a final hour are passed. It is also possible to indicate whether both hours should be included in the filter, being activated by default.

For this function it has been necessary to check if the hour formats passed by parameter were correct. These might be:

```
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' ]
```

Algorithm 7.5: Accepted time formats for filtering in *Track Animation*

being *H* and *I* the hours in 24 hour and 12 hour format, respectively, *M* the minutes, *S*
the seconds and *p* to indicate whether it is AM or PM.

After checking the formats, a variable of the type has been created *DatetimeIndex* with the values of the date column of the trajectories to be able to use the function *indexer_between_time.* The returned values are filtered into the dataframe by the function *iloc,* which is nothing more than a selector by position using booleans.

C CHAPTER 8

V ISUALIZATION OF INDICATORS FROM OF COLORS ON THE TRAJECTORS

Indicators are essential to be able to analyze and interpret large amounts of data in an efficient and simple way. Given the *Track Animation* It only exports visualizations of the trajectories, the function has been implemented *setColors* to be able to see the values of the indicators by applying a color palette to the paths. Altitude, speed, time traveled, or any other custom indicator that a user creates can be turned into a gradient that becomes visible when a video, image, or interactive map is exported.

The indicators must have a value for each point of the trajectory. The color applied to each point is calculated based on the maximum and minimum value of the indicator. Thus, in table 8.1 you can see an example of which are the RGB color codes that are assigned to an indicator that goes from 1 to 3. As can be seen, the RGB values go from 0 to 255 However, the bookstore *matplotlib* It only admits values between 0 and 1 to paint them on a figure.

Value	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	fifty	205	0

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

Table 8.1: Example of assigning colors to an indicator

The color palette used for any indicator is the one shown in figure 8.1, with blue being the lowest value of the indicator and red being the highest value.

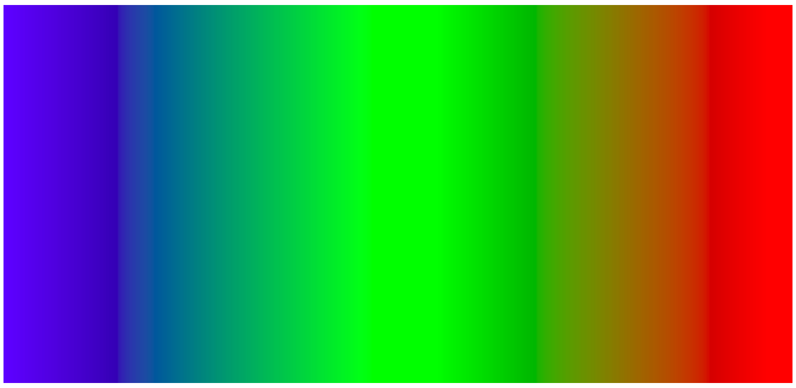
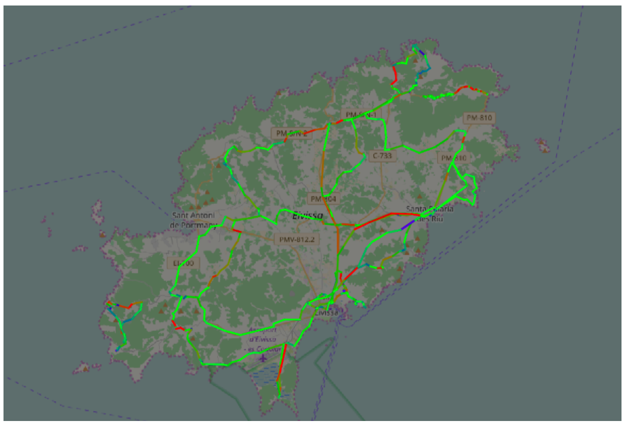
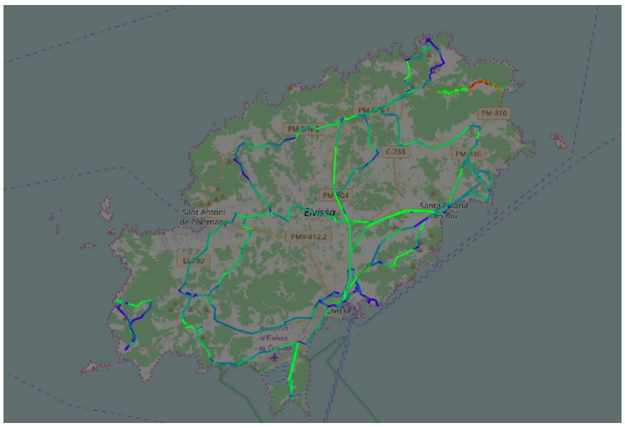


Figure 8.1: Color palette used by the trajectory indicators

To allow a user to further customize the displays and compare between paths, it is possible to specify whether to apply the color palette to each path individually or to all paths together. This is indicated by the parameter *individual_tracks* of the function *setColors,* which defaults to true. An example of this can be seen in figure 8.2. As can be seen, image a) has colors that go from blue to red in each of the trajectories, being able to see in which sections of each trajectory it has gone faster or slower. However, in image b) the colors are applied to all the trajectories, making it possible to see which of them has been the fastest, for example.



(a) By trajectory



(b) Joint trajectories

Figure 8.2: Comparison of the speedometer color gradient applied to each trajectory individually and as a whole

C CHAPTER 9

V TRAJECTORY UPDATE PROCESSED IN *Track Animation*

The main objective of *Track Animation* is to generate visualizations of GPS tracks stored in GPX format files. This part is what causes the trajectories to be easily interpretable from the handling that has been made of the data, either by specifying the seconds of duration or the number of FPS of a video, on which indicator the colors are to be created , which places have to be seen or which points have to be discarded.

All this processing can be seen reflected in three different types of visualizations: *videos, images* and *interactive maps* from OpenStreetMap. For this, the library is used *matplotlib* that generates a figure of one or more graphs with all the points of one or more dataframes of the class *DFTrack.* An algorithm has been designed that manages the way in which the points of the different trajectories are visualized in order that each one of them is independent of the other and generates the same route that was made in reality. However, for large amounts of data and for visualizations that are static or do not require frames, such as an interactive map, another algorithm has been designed that captures the complete trajectories in the fi gure, without going point to point.

FFmpeg [19] is the tool used to generate videos from sequences of images created by *matplotlib* based on the number of FPS specified by the user. The higher the number of FPS, the faster the trajectory visualization is in case there has not been a previous normalization of the points that compose them [Chapter 6]. The library has been used *mpllea fl et [* 38] to facilitate the visualization of trajectories in an interactive OpenStreetMap map [Figure 9.1].

For the implementation of everything related to visualizations, the class has been created *AnimationTrack* inside the module *animation.py.* The main object of this class is the figure created with *matplotlib,* which has a graph for each *DFTrack* passed by parameter. Points are added to these graphs to finally generate one of the three types of visualizations.

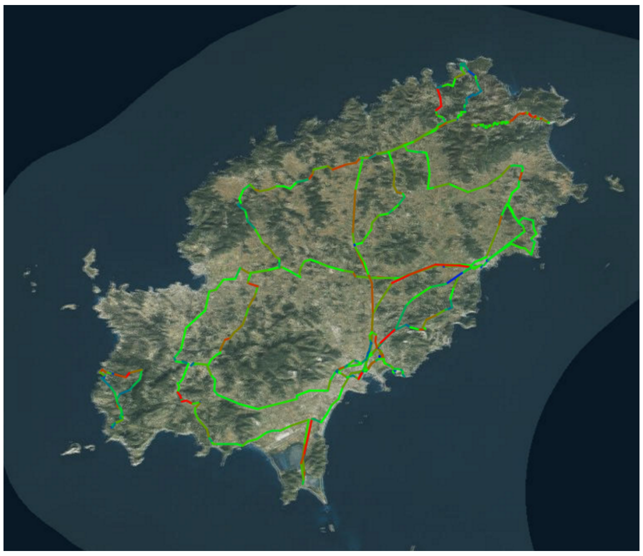


Figure 9.1: Ibiza cycling paths on an interactive OpenStreetMap map

This last chapter is made up of five sections. The first gives an introduction to the library *matplotlib* to explain what a figure is and what parts it is made of. The second section details in depth how the management of points and trajectories is carried out when you want to create a visualization, explaining the data structure implemented so that the export is carried out as quickly as possible with the largest amount of data. that is desired. Sections three, four, and five explain each of the visualizations that can be created in *Track Animation:* videos, images and interactive maps, respectively.

9.1 Using the library *matplotlib* in *Track Animation*

matplotlib is a library that aims to create data visualization figures and graphs quickly and easily. This library is the basis with which all the visualizations have been implemented due to its ease of customization and integration with other libraries.

the basis of *matplotlib* are the figures, from which the other components that make up the data visualization are created 9.2. These components are the *Axes,* titles, legends, *canvas,* etc. The *canvas* They are useful for drawing the fi nal drawing of the entire fi gure when it is rendered, but should be practically invisible to the user. The *Axes* is a region of the figure that contains the data. A figure can contain several *Axes* and these are formed by two axes (***Axis),*** which are the limits of the data. These axes are responsible for creating the ***ticks,*** that is, the marks on them and their respective labels.

The easiest way to create a new figure is as follows:

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

Algorithm 9.1: Basic creation of a figure in *matplotlib*

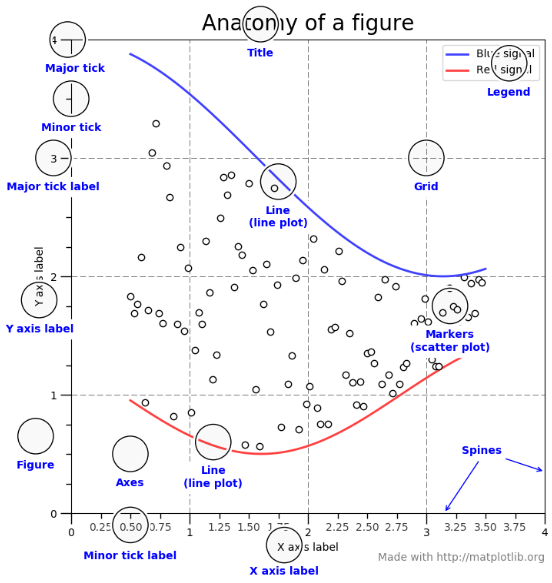


Figure 9.2: Parts of a figure of *matplotlib [* 1]

In the following example, the first call to the function *plt.plot* is in charge of creating the figure, which automatically generates its *Axes* with the necessary measures. The following calls to this function use the *Axes* already created adding only new lines.

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace (0, 2, 100)

plt.plot (x, x, label = 'linear' )
plt.plot (x, x ** 2, label = 'quadratic' )
plt.plot (x, x ** 3, label = 'cubic' )

plt.show ()
```

Algorithm 9.2: Example of use of *matplotlib*

When instantiating the class *AnimationTrack* of *Track Animation,* a new figure is created with a number of *Axes* based on the number of objects *DFTrack* passed by parameter, that is, several sets of trajectories that are displayed independently within the same fi gure can be passed. The limits of these are those indicated by the maximum and minimum coordinates of the data of each *DFTrack,* using function

getBounds, so that the display is centered on the GPS points they contain. You can also specify whether you want to view the display with a background map of *OpenStreetMap.*

```
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 = 'car' )
        else :
            self . axarr [i] .set_ylim ([min_lat, max_lat])
            self . axarr [i] .set_xlim ([min_lng, max_lng])
```

Algorithm 9.3: Simplified __init__ of the class *AnimationTrack*

9.2 Management of points and trajectories for visualizations

Two functions have been created to manage the way in which each point and each trajectory is visualized on a figure of *matplotlib.* The first of them, **computePoints,**
It traces all the trajectories point by point, being very useful for any visualization that requires the use of frame rate, such as, for example, videos or the printing of an image in a specific second. The second function, **computeTracks,** takes care of plotting the complete trajectories, so you have to use the function *plot* a very reduced number of times compared to the previous function, being very useful for exporting images or interactive maps of the entire trajectories.

Point-to-point tracing in the function *computePoints* It is necessary for each trajectory to be displayed individually on a figure of *matplotlib* calling function *plot* for each new point. Two problems arise with this approach:

1. It is not possible to add a new point to an already created line, so that line must be deleted and recreated with the new point.

2. It is not possible to paint two different lines with just one *plot,* so they can't plot two points of two different trajectories even though these have been generated

at the same time or have to go in the same frame.

For this reason a data structure has been created consisting of a Python dictionary, *track_points,* It contains the points already drawn for each trajectory. Each position in this dictionary is a path that another dictionary has, *position,* with only two elements: the latitude and the longitude of each point. These two elements are lists of coordinates that are passed to *matplotlib* to do a *plot* of the points already drawn on a specific path [Figure 9.3]. Paths and points are added to this data structure as the dataframe is iterated.

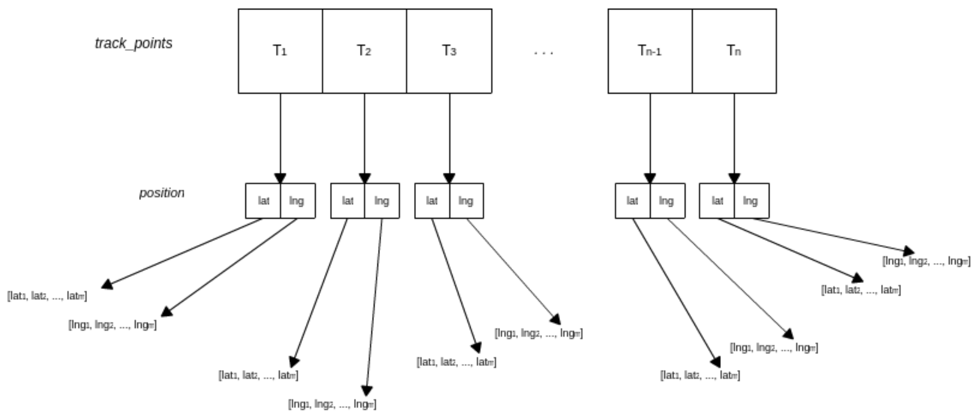

track_points
T_1
T_2
T_3
. . .
T_{n-1}
T_n
position
lat
lng
[lat_1, lat_2, ..., lat_m]
[lng_1, lng_2, ..., lng_m]

Figure 9.3: Implemented data structure for point-to-trace management point in visualizations

The function *computePoints* It consists of a main loop that goes through each of the points of the dataframe and adds them to the data structure to make the *plot.* Since the computation of this algorithm is point-to-point, the function creates a generator that returns, in each iteration, the point drawn at that moment and the next one in the dataframe. This is done using the keyword *yield* instead of *return.*

Generators are iterators that can only be used once since they do not store all values in memory, but rather generate them at runtime. The first time the generator is called in a *for,* the function code runs from scratch until it finds the *yield* in the main loop, returning the point you have plotted at that time and the next point in the dataframe. In the next iteration of the generator, the function's loop is executed again, returning the following points.

```python
def computePoints ( self , linewidth = 0.5):
        track_points = {}

        points = self . track_df.toDict ()

        # Main loop of the function
        for point, next_point in zip_longest (points, points [1:], fillvalue = None):
                track_code = point [ 'CodeRoute' ] + "_" + point [ 'Axes' ]

                # Check if the track is in the data structure
        if track_code in track_points:
            position = track_points [track_code]
        else :
            position = {}
            position [ 'The t' ] = []
            position [ 'lng' ] = []

        lat = point [ 'Latitude' ]
        lng = point [ 'Longitude' ]
        if self . map :
            lng, lat = self . map [ int ( point [ 'Axes' ])]. to_pixels (lat, lng)

        position [ 'The t' ] .append (lat)
        position [ 'lng' ] .append (lng)
        track_points [track_code] = position

        self . axarr [ int ( point [ 'Axes' ])]. plot (position [ 'lng' ], position [ 'The t' ], lw
            = linewidth)

        yield point, next_point
```

Algorithm 9.4: Simplified point-to-point trace management algorithm

The algorithm *computeTracks* is much simpler, since it only makes a *plot* for each of the trajectories in the dataframe. This is due to the fact that to represent a final image of the trajectories it is not necessary to compute them point to point, but simply pass the function *plot* the list with its coordinates.

9.3 Video generation

The generation of videos with the imported trajectories is one of the essential requirements for which the library has been created *Track Animation.* However, implementing an optimal algorithm that computes a large number of points and generates the necessary frames to create a video has been a complex task. During the early stages of its development, the method was used *save fi g* from the bookstore *matplotlib* to generate an image in *png* of each frame and, later, join them using FFmpeg. In this way, the creation of a 691 point video took around 17 minutes because the generation of each *png* it is an output to hard disk that interrupts the execution of the algorithm. In addition, the function *save fi g* performs other tasks and checks before saving each image. Another reason is that an image was generated for each of the points in the dataframe, even if they coincided in time. At that time, the normalization function had not yet been implemented [Chapter 6].

Normalization has allowed us to know the number of frames that are created and, in addition, that it is not necessary to generate an image for each of the points of the data frame. When the algorithm finds a collection of points that should go in the same frame, makes a *plot* of each of them, as explained in the previous section, but it generates a single image. Also, the function is implemented in such a way that it saves each image in a *buffer* main memory and passes its information to a pipe (*pipe)* for FFmpeg to create the video at the same time as the algorithm runs. So, normalizing the 691 points mentioned above so that the video lasts 10 seconds at 10 FPS, 1028 points come out, creating the video in 21 seconds.

This section is intended to explain in two subsections what has been the union that has been made of FFmpeg, the algorithm *computePoints* and the *buffers* memory to make video generation faster and more efficient.

9.3.1 Using the FFmpeg library

FFmpeg is the library used to generate the videos from the sequences of images created. It is free software and allows the manipulation, creation and conversion, among other operations, of audio and video. In *Track Animation* FFmpeg is used in parallel to the execution of the algorithm that manages the plotting of the points for a higher speed, sending the images created through a pipe (*pipe).*

```
cmdstring = ( 'ffmpeg' ,
          ' -and' ,
          ' -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)
```

Algorithm 9.5: Pipeline creation (*pipe)* to generate the videos with FFmpeg

The arguments used from FFmpeg are the following:

- **- and:** an existing file with the same name is overwritten.
- **- loglevel:** speci fi es the level of information displayed on the screen. The value *quiet* does not display any information.
- **- framerate:** indicates the number of input images that pass through the pipe to generate one second of video.
- **- f:** forces the input or output to be a speci fi c shape. In this case, since it is located before the input (- *i),* force it through the pipe.
- **- i:** indicates that the inlet is through the pipe.

- **- r:** specifies the number of frames per second in the final video. The difference it has with - *framerate* is that, being indicated for the output, it duplicates or deletes frames that have entered depending on whether their value is higher or lower. It has been used so that all videos are seen with higher quality [Section 6.2] regardless of the number of frames that is specified as input, being totally independent of this since it only duplicates or deletes the frames already generated.

- **- s:** indicates the size of the fi nal video.

- **- pix_fmt:** indicates the pixel format. By default, for creating videos *. mp4,* FFmpeg puts it at *yuv444,* which is not compatible with the majority of players that exist [46]. For this reason, it has been specified *yuv420p.*

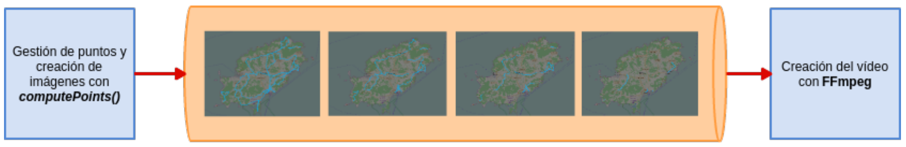

Gestión de puntos y creación de imágenes con ***computePoints()***
Creación del video con **FFmpeg**

Figure 9.4: Pipe (*pipe)* through which the images of the *buffer* from memory to FFmpeg

9.3.2 Use of *Buffers* memory in the process of creating frames of a video

Once the pipeline is created to pass the images to FFmpeg, a loop is created with the function *computePoints.* It should be remembered that this function creates a generator that returns, in each iteration, the point drawn at that moment and the next one in the dataframe. Thanks to this and, making use of the column *VideoFrame* of the dataframe, it is possible to check if the next point is in the same frame as the current point to generate only the necessary frames and make the video last the desired time in case there has been a previous normalization of the trajectories.

The bookstore *matplotlib* has the function implemented *save fi g,* which saves the fi gure as it is at the time of your call. This function performs other tasks and checks before saving the fi gure, which may take additional time to the total generation of the video. For this reason, a way has been implemented to save the images created in main memory using the *canvas* of the figures and the module *Image* from the bookstore *PIL* to convert them to PNG format. Created PNG images are saved in a *buffer* memory that is later passed to the pipeline.

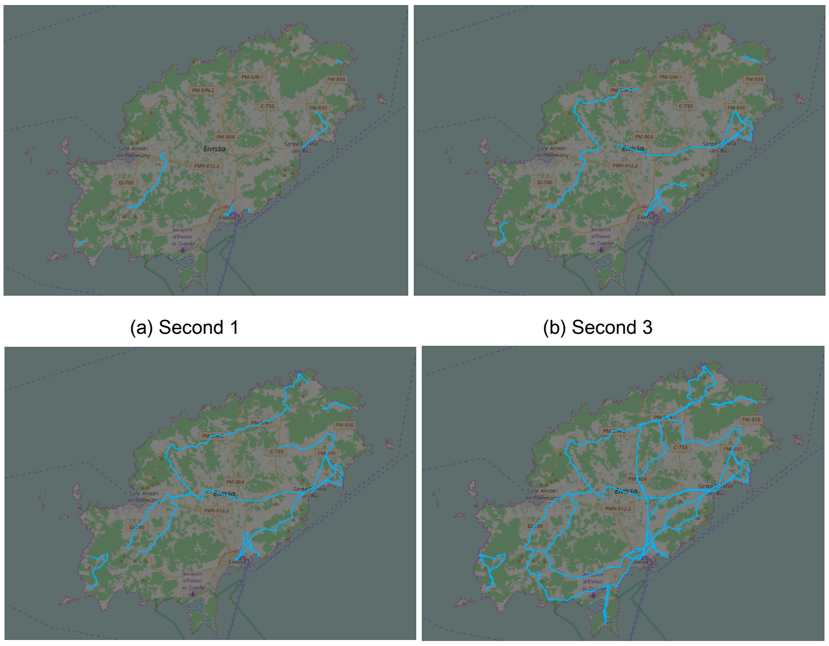
```
for point, next_point in self . computePoints (linewidth = linewidth):
    if self . isNewFrame (point, next_point):
        buffer = io.BytesIO ()
        canvas = plt.get_current_fig_manager (). canvas
        canvas.draw ()
        pil_image = Image.frombytes ( 'RGB' , canvas.get_width_height (), canvas. tostring_rgb ())
        pil_image.save ( buffer , 'PNG' )
        buffer . seek (0)
        pipe.stdin.write ( buffer . read ())
```

Algorithm 9.6: Creating the frames of a video

As can be seen from the algorithm, it is first checked by the current point and the next if a new frame is to be created. If not, the algorithm goes to the next point. If so, the *buffer* of memory in which the images generated using the module are saved *io* Python. Using the function *draw* of *canvas* of *matplotlib* the figure is drawn to later convert the pixels into an image using the function *frombytes* from the bookstore *PIL.* Once the image is generated, it is saved in the *buffer* in PNG format, it points to the first position of this and the information is passed to the pipeline.

9.4 Generation of images

The objective of *Track Animation* It is to create different types of visualizations of the introduced trajectories in order to be able to interpret them in a more friendly and simple way. For this reason, it is also possible to generate images of the trajectories, either an image in which all of them have ended as one or more images of specific seconds that are specified by parameter. It should be mentioned that, if it is previously normalized, the duration of a video with a specific number of frames is known and, therefore, this can be used to take an image of how the trajectories would be in a given second.



(a) Second 1

(b) Second 3

(c) Second 5

(d) Second 10

Figure 9.5: Sequences of images per second

For the generation of images the function *makeImage.* This function can be specified by parameter the FPS with which the trajectories have been normalized and a second or a list with seconds in which you want to create an image. In the case of not specifying any second, it is understood that the only thing that the user wants is to take the final image of all the trajectories, so the function used for this purpose is that of *computeTracks,* since it is not necessary to compute point to point. If seconds are specified, use the function *computePoints*

to calculate point by point which second is being processed until it matches one of those specified, at which point the function is used *save fi g* of *matplotlib*
to generate the image. In this case, the function is used *save fi g* because a sequence of images does not have to be generated and, therefore, the computational cost of this is imperceptible.

9.5 Generation of interactive maps

Other types of visualizations that can be created with *Track Animation* are interactive maps generated by the library *mpllea fl et,* which is capable of superimposing a figure of *matplotlib* on an OpenStreetMap map. With this type of visualization it is possible to zoom out or approach the trajectories and move around the map 9.6.

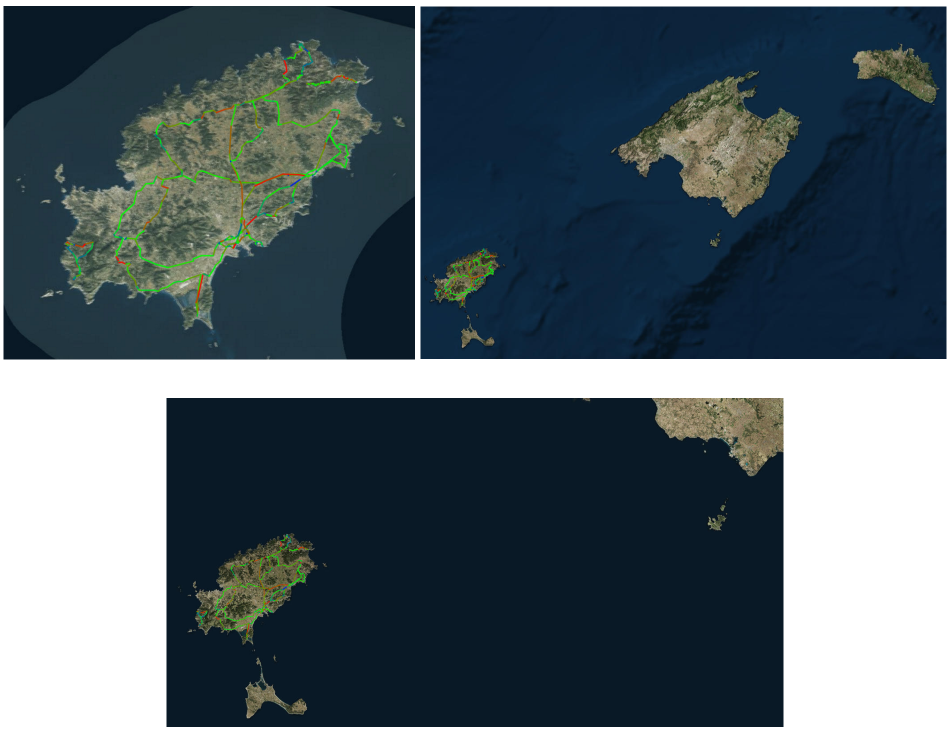


Figure 9.6: Interactive map with different zoom levels

The function implemented for this purpose is *makeMap,* which uses only *computeTracks* because it is not necessary to go point to point. Finally, the function *save_html* of *mplleaflet* creates an HTML file with the interactive map and the paths processed on top of it.

C CHAPTER 10

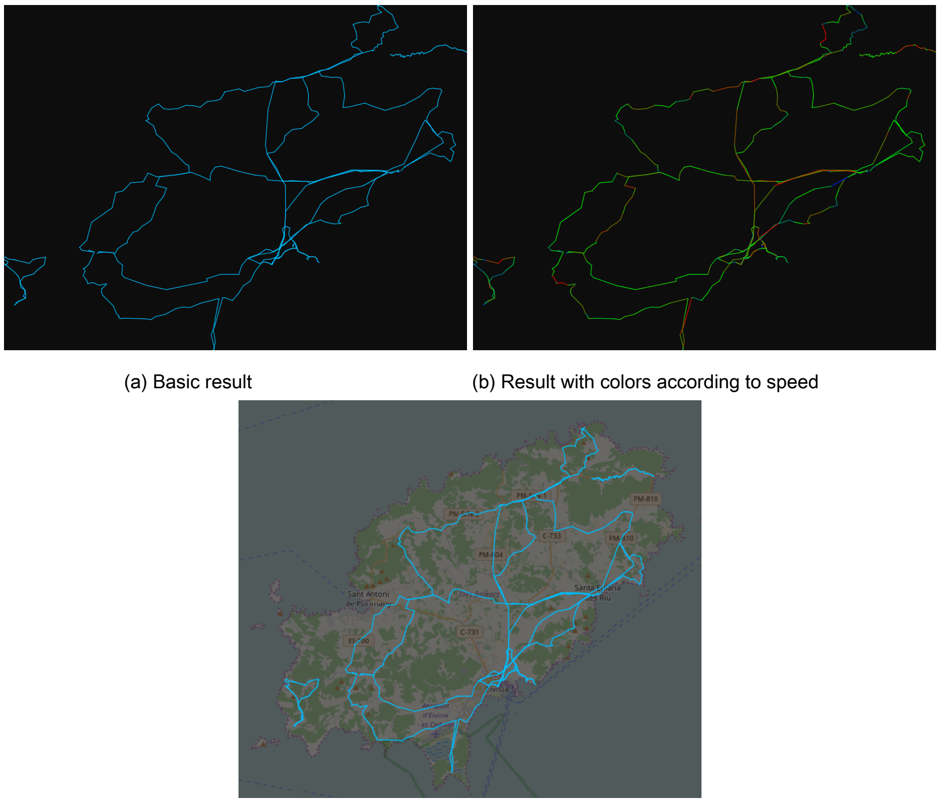
C ONCLUSION

In this document I have explained how it is made and how the library is used *Track Animation* Python. The main objective of *Track Animation* is to generate visualizations of GPS trajectories stored in files with GPX format in a way in which any end user can avoid problems and technical constraints to use it. This has been achieved thanks to the meticulous implementation of an API with which it is possible to manipulate and process GPS trajectory data to later export a video, an image or an interactive map. Without any library that provided this type of support, all these processes were expensive.

A whole set of guidelines has been provided for using the library *Track Animation,* first explaining its architecture [Chapter 3], followed by its most basic functions [Chapter 5] and ending with the set of main functionalities that allow data manipulation and processing [Chapters 6 - 9]. The latter are the normalization of the GPS trajectories, the filters applied to them by date, time and place, the application of colors to each of their points based on the values of an indicator and the final export of each one. of the visualizations. Several third-party libraries have been used to facilitate the implementation of the set of *Track Animation,* among which stand out *gpxpy [* 31] for reading and parsing GPX files, *pandas [* twenty] for manipulation and processing of trajectories using dataframes, *matplotlib [* 22] With *smopy [* 37] to generate the visualizations of the trajectories and add background maps and *geopy [* 36] to obtain the coordinates of places to be able to filter the trajectories.

The GPX and CSV files are the two input formats of the library *Track Animation.* It has been observed that reading 1000 GPX files of an average of 150 kB (more than a million points) takes around 3 minutes, while reading the exact same points saved in a 136 MB CSV file takes 5 seconds. This is because the CSV does not have to parse the tags that make up the XML. The points read from the files are passed to a dataframe in the library *pandas* so that they can be processed before creating a visualization.

The normalization of trajectories is one of the most important pillars that



(a) Basic result (b) Result with colors according to speed

(c) Basic result with background map

Figure 10.1: Example of different visualizations that can be created with *Track Animation*

make up the project. It allows you to lengthen or shorten the trajectories so that they all start and end at the same time, regardless of their duration or distance traveled. In addition, it allows you to specify the exact time you want a video to last with those trajectories imported to *Track Animation,* adding new points if necessary to give an effect of continuity and uniformity to these when viewed. This makes the frames per second have a great importance on the result of this function and, consequently, the exported video, since it allows to shorten the time between two consecutive points. In other words, if the FPS had not been taken into account, the trajectories would be seen giving jumps of different sizes depending on the difference in which the GPS captured each of the points.

Other library functionality *Track Animation* It is the filtering of the trajectories based on date, time or place, thus allowing the creation of different sets of trajectories with a smaller number of points that make the final interpretation easier on a visualization [Figure 10.2]. For this, the library has been used

geopy, to obtain the coordinates of the place through which a user wants to filter the trajectories, and the time series of *pandas* to filter by date and time.

The objective of *Track Animation* is to create visualizations of the imported trajectories and, for this, it is possible to export videos, images and interactive maps. It makes use of *matplotlib* to create figures with the points of the trajectories that, later

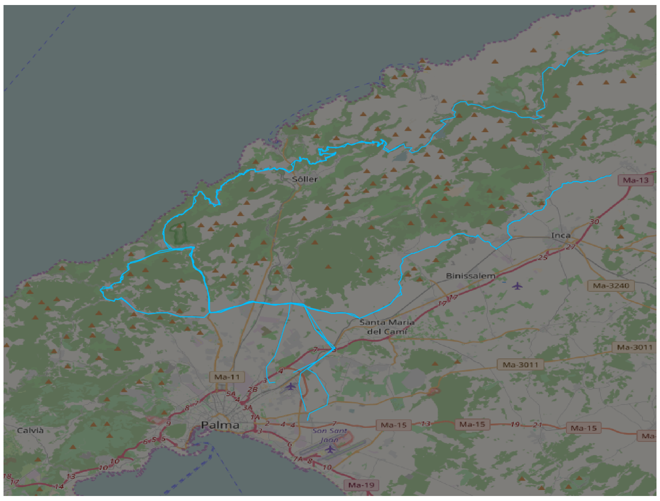


Figure 10.2: Trajectories that pass through Valldemossa between 08:00 and 12:00 first semester of the year from 01/01/2010 to 06/01/2014

Mind you, they are saved in one of the 3 formats discussed. With *FFmpeg [* 19] videos are generated from the images that are created and stored in a *buffer* memory, making them arrive through a pipe (*pipe)* so that it is totally independent of the point management carried out by the implemented algorithm. This is because saving each image generated for each frame of the video to disk requires a high output time of information and one interruption after another of the algorithm. When doing it in parallel, with the *buffer* memory and also using the normalization function, a 691-point video can be exported in 21 seconds, while saving each image to disk requires approximately 17 minutes.

10.1 Future work

Due to the large volume of GPS data that is generated today and the need for applications to view them in a simple and fast way, *Track Animation*
it is a project that has to continue and, in addition, be reproduced on other platforms. There are already several functionalities designed to be implemented in future versions such as, for example, reading from other data sources, the export of images with geopositioned information that can be imported back to *Track Animation,*
the creation of more personalized maps with different styles [47], obtaining the climatic time that was when the trajectory was being made [48] or the addition of labels with information about the trajectory in the visualizations.

In addition, as the library is implemented and structured, it is easily reproducible to other types of applications. For example, the implementation of a plugin for the QGIS program [28] with the aim that it is possible to interact with the trajectories using other third-party plugins at the same time. Another example is the creation of a web page with Django where any user can upload their GPS data, interact and manipulate the trajectories through an interface graph and, later, you can see the videos generated on the website itself, download them or share them with other people.

Track Animation It is a free software library and anyone can contribute new ideas and implementations in order to make it grow more and more.

10.2 Personal opinion

Track Animation It is a project that I am truly proud of for being something different from what students are used to doing in computer engineering. The Python programming language, the creation of a library from scratch using and joining others, the manipulation of GPS data or the conversion of said data to an animated visual format, such as video or image, is something that never had done and, if it were not for this project, perhaps I would not have even considered doing. I have enjoyed a lot throughout the project, being totally meticulous and perfectionist in everything I did in order to fulfill each of the objectives that I had set at the beginning. Also, having learned to use Python and the library *pandas* has caused me to start working in the department of *Big data* of the company that I am currently in, which is a great goal in my life because it is something I know I have wanted to dedicate myself to for a few years.

Each of the objectives that were raised at the beginning of this project have been covered. *Track Animation* It is a project that can continue and grow, since there are not a large number of applications that dedicate so much effort so that anyone can use them in such a simple and customizable way as this one does. In this way, the essence of the project is that the union of several libraries, such as *pandas* or *matplotlib,* make any analysis that is being carried out of GPS tracks can be imported into *Track Animation* to finish generating a visualization.

I have not done this project thinking that it would only be a final degree project, but thinking that it is something that may have a future, that it will be totally public and that anyone will be able to use. It is for this reason that, despite having fun, there have been times when I even border on obsession and invested a great amount of time in continuing to improve parts that were already implemented while creating new functionalities, and for the simple fact of wanting whoever uses the library to do so at ease and not think of looking for another that better meets their requirements. However, I am frankly happy to have implemented a library to which, with small changes and in a very short time, new functionalities can be added, since there is a very good base created.

B IBLIOGRAPHY

[1] "Matplotlib usage." [On-line]. 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 ZX 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, AK 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. [On-line]. Available: http://vespapp.uib.es/ 1

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

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

[9] S. Khetarpaul, R. Chauhan, SK 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 andM. Mamei, "Identifying and understanding urban sport areas using nokia sports tracker," 2013. 1

[eleven] Garmin. [On-line]. Available: http://www.garmin.com/es-ES/ 1, 2.3 Tomtom. [On-line].

[12] 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.topogra fi x.com/gpx.asp 1, 1, 2.2, 4

[15] Bikely. [On-line]. Available: http://www.bikely.com/ 1 [16] Wikiloc. [On-line].

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. [On-line]. Available: https://play.google.com/store/apps/details?id= com.vecturagames.android.app.gpxviewer & hl = en 1, 2.3

[19] Ffmpeg. [On-line]. Available: https://ffmpeg.org/ 7, 7, 9, 10 [20] Pandas. [On-line].

Available: http://pandas.pydata.org/ 1, 5, 3.2, 4, 10

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

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

[23] E. Rampinini, G. Alberti, M. Fiorenza, M. Riggio, R. Sassi, TO Borges, and AJ Coutts, “Accuracy of gps devices for measuring high-intensity running in fi eld-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, "The 10 formats chalk vector plus populares, " 2017. [On-line]. Available: https://mappinggis.com/2013/11/ the-most-popular-vector-chalk-formats / 2

[26] Gpx documentation. [On-line]. Available: http: //www.topogra fi x.com/GPX/1/1/ 2.1

[27] Nasa world wind. [On-line]. Available: https://worldwind.arc.nasa.gov/ 2.3 [28] Qgis. [On-line].

Available: https://www.qgis.org/es/site/index.html 2.3, 10.1 [29] igo navigation. [On-line]. Available: https://www.igonavigation.com/

2.3 [30] Track road. [On-line]. Available: http://www.trackroad.com/ 2.3 [31] gpxpy. [On-line]. Available: https://github.com/tkrajina/gpxp

2.3, 1, 3.2, 4, 10 [32] Gpx aviation. [On-line]. Available: http://navaid.com/GPX/ 2.3 [33] Bbbike. [On-line].

Available: http://www.bbbike.de/cgi-bin/bbbike.cgi 2.3

[34] Gps tracks. [On-line]. Available: http://www.gps-tracks.com/ 2.3 [35] Trace gps. [On-line]. Available: http://www.tracegps.com/

2.3 [36] Geopy. [On-line]. Available: https://github.com/geopy/geopy 5, 3.2, 10 [37] Smopy. [On-line].

Available: https://github.com/rossant/smopy 7, 3.2, 10 [38] mpllea fl et. [On-line]. Available: https://github.com/jwass/mpllea

fl et 7, 3.2, 9 [39] tqdm. [On-line]. Available: https://pypi.python.org/pypi/tqdm 3.1, 3.2 [40] Image.

[On-line]. Available: http://effbot.org/imagingbook/image.htm 3.2

[41] BF Janzen and RJ 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] LM Wilcox, RS Allison, J. Helliker, B. Dunk, and RC Anthony, “Evidence that viewers prefer higher frame-rate fi lm,” 2015. [Online]. Available: https://static1.squarespace.com/static/565e05cee4b01c8 57ae0ce42994ca2ad32bf74f / 1471024357823 / WilcoxAllisonetal + ACM + 2015.pdf 6.2

[43] QL 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. [On-line]. Available: https://github.com/lforet/robomow/ blob / master / navigation / gps / gps_functions.py # L223-L240 6.3

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

[46] H.264. [On-line]. Available: https://trac.ffmpeg.org/wiki/Encode/H.264 9.3.1 [47] Folium. [On-line].

Available: https://github.com/python-visualization/folium 10.1

[48] Wunderground. [On-line]. Available: https://www.wunderground.com/weather/ api 10.1