UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Degree in Computer science

DEGREE WORK

scikit-fda: Interactive Visualization and Analysis Tools for Functional Data

Author: Álvaro Sánchez Romero
Advisor: Alberto Suárez González

May 2021

Álvaro Sánchez Romero
scikit-fda: Interactive Visualization and Analysis Tools for Functional Data

Álvaro Sánchez Romero
C\ Francisco Tomás y Valiente N$^{\underline{o}}$ 11

PRINTED IN SPAIN

A mi madre por apoyarme siempre incondicionalmente

Necesitamos enseñar a que la duda no
sea temida, sino bienvenida y debatida.
No hay problema en decir: 'No lo sé'.

Richard Feynman

Prefacio

Este Trabajo de Fin de Grado ha sido creado con el propósito de extender una de las primeras librerías de datos funcionales en Python, scikit-fda. Esta librería está integrada en el ecosistema SciKit que incluye paquetes Python para matemáticas, ciencia e ingeniería. En concreto, se han diseñado e implementado herramientas interactivas para la visualización y análisis de datos funcionales. Los datos funcionales corresponden a curvas, superficies, etc. cuyo valor depende de un parámetro continuo. Para su análisis partimos de una muestra compuesta por un conjunto de funciones, que pueden ser consideradas como una realización independiente de un proceso estocástico. Debido a estas características es necesario desarrollar métodos de análisis estadístico y visualización más complejos que los utilizados en la estadística multivariante.

Durante su desarollo se ha tratado de facilitar al máximo el uso de esta parte de la librería, gracias a la estandarización, homogenización y reestructuración de la funcionalidad de visualización. Adicionalmente, se han completado y extendido herramientas de visualización y análisis fundamentales para la caracterización de los datos funcionales. Por último, se ha dotado de interactividad a los gráficos, que hacen posible la exploración dinámica de los datos. Gracias a esta funcionalidad el usuario dispone de herramientas avanzadas para extraer el máximo de información de los datos.

Como autor del trabajo, espero que este documento les sea entretenido y lúdico, tal y como me sucedió a mí durante el desarrollo del mismo.

Agradecimientos

Quiero dar gracias a todo el grupo de trabajo que conforma este gran proyecto, scikit-fda. A Carlos Ramos, por su gran ayuda tanto con la librería como con el mundo de los datos funcionales. A mi tutor, Alberto Suárez, por su constante trabajo y disposición a orientarme al seleccionar el proyecto. A mis compañeros, Elena Petrunina y Pedro Martín, que me han acompañado virtualmente a lo largo de la realización del proyecto. También quiero dar gracias al profesorado de la universidad, los cuales me han ayudado a llegar donde estoy, especialmente a Eloy Anguiano, por su plantilla para la realización de este documento. Por último, quiero dar las gracias a mi familia por apoyarme siempre que lo he necesitado.

Resumen

En este trabajo se han desarrollado herramientas interactivas para la visualización y análisis de datos funcionales en la librería *scikit-fda* [1]. La librería scikit-fda para el análisis de datos funcionales está integrada en SciPy [2], un ecosistema de código abierto para matemáticas, ciencia e ingeniería, desarrollado en Python. El Análisis de Datos Funcionales es la rama de la estadística que estudia variables aleatorias que dependen de un parametro continuo; es decir, funciones aleatorias como curvas, superficies, etc. Este campo por ejemplo podría estudiar el crecimiento de la altura de una persona durante su juventud (en este caso el continuo sería el tiempo). Para poder obtener información gráfica de estos datos se han ido desarrollando gráficas que permitan obtener detalles sobre las curvas, valores atípicos, parametrización de funciones, entre otros.

Para empezar, se realizará un análisis comparativo de las librerías de visualización y análisis de datos disponibles. En este estudio se prestará especial atención a las herramientas interactivas que proporcionan al usuario. Con el fin de establecer el contexto en el que ha sido desarrollado el proyecto, se proporcionará una somera descripción de la estructura y funcionalidad de la librería *scikit-fda*. Después se discutirá sobre las principales herramientas de visualización desarrolladas: un detector de curvas con forma atípica, un método para parametrizar funciones o un gráfico que permite comparar un conjunto de datos con dos distribuciones. Adicionalmente, también se ha implementado un módulo que permite interaccionar con todas las gráficas mediante widgets o con el propio cursor. Para esto ha sido necesario investigar las mejores y más eficaces soluciones que permitan que la herramienta funcione en todas las interfaces gráficas de usuario (GUI).

También el documento explicará las etapas de desarrollo software seguidas para conseguir diseñar e implementar el software. Se hablará sobre la documentación, tipos de test y mecanismos de trabajo utilizados. GitHub es la herramienta seleccionada para compartir los progresos y donde está disponible el código [3].

Mis objetivos como trabajo de fin de grado son los de lograr crear un conjunto de herramientas de visualización que ayuden a los usuarios, además de crear una interfaz más fácil, estándar y homogénea para los métodos ya existentes y los nuevos.

Palabras clave

Análisis de Datos Funcionales, visualización, interactividad, Python, Matplotlib, scikit-fda (Paquete de Python para FDA), medidas de profundidad, software de código abierto

Abstract

In this work interactive tools have been developed for the visualization and analysis of functional data in the *scikit-fda* library [1]. The scikit-fda library for the analysis of functional data is integrated in SciPy [2], an open-source ecosystem for mathematics, science and engineering, developed in Python. Functional Data Analysis (FDA) is the branch of statistics that studies random variables that depend on a continuous parameter; or what is the same, random functions like curves or surfaces. FDA could for example study the growth on the height of a person during his youth (in this case being the continuum time). To be able to obtain graphical information from this data, new graphics had being developed to obtain details of curves, atypical values, parameterization of functions and others.

To start, a comparative analysis of the visualization and data analysis libraries available will be done. During this study attention will be paid to the interactive tools that are given to the user. With the objective of establishing the context in which the project has been developed, a brief description of the structure and functionality of *scikit-fda* will be given. After that, there will be a discussion about the main visualization tools developed: a shape outlier detector, a method to parameterize functions and a graphic that allows to compare a sample of data with two distributions. Besides that, I have also worked in a module that allows you to interact with all the different graphics thanks to widgets or with your own mouse. For this it has been necessary to investigate the best solutions and the most efficient ones that allowed the plots to work in every graphical user interface (GUI).

Furthermore, this document will talk about the software steps that had to be followed to manage to design and implement the project. It will comment on the documentation, types of tests used and work mechanisms followed. GitHub is the tool used to share our progress and where the code is currently available for everybody [3].

My objectives with this degree work are creating a set of visualization methods that help the users, besides creating an easier, more standard and homogeneous interface for the already existing methods and the new ones.

Keywords

Functional Data Analysis, visualization, interactivity, Python, Matplotlib, scikit-fda (Python package for FDA), depth measures, open-source software

Table of Contents

Lists

List of figures

1

Introduction

Over the years, new fields in statistics had been growing. One of them is Functional Data Analysis (FDA). This area's object of study are data consisting of curves, surfaces or any quantity that changes over a continuum. An example of these types of data is a collection of functions that depend on variables such as time or space. Even if its beginnings are in the 1940s, core advances were made in recent time thanks to Ramsay and Silverman [4] and Ferraty and Vieu [5]. These are some of the fundamental references of this project as they have helped me to understand the most important concepts of FDA.

One of the interesting aspects of this branch of statistics, is the multiple areas it can be applied to, like financial time-series, biomedical signal, climate patterns, and so on. As interest grows, more frameworks appear related with this topic. Most of these are coded in the programming language, R; even if we can find some in Matlab. In this language we can find multiple packages such as fda.usc [6], fda [7] (which was also developed by Ramsay), fdasrvf [8] and others. Some of these libraries have a more general objective while others are more specialized in certain functionalities like regression, classification or clustering. Almost all the packages in R related with this matter, are stored in the repository CRAN, The Comprehensive R Archive Network [9].

A functional dataset can be very simple to represent graphically. For example, a set of curves in a 2 dimensional space. The representation of surfaces or higher dimensional-quantities poses more difficulties. For this reason, it is important to have a good visualization module that allows users carry out a visual exploration that increases their understanding of the data. The objective is to create a simple and homogeneous interface for the user in which he can visualize easily his plot.

In this project I collaborate on the development of the visualization and interactivity module of a FDA package in Python, *scikit-fda* [1]. This package is an open-source software project started in 2018 and in which many people have contributed [10]. After that, it has been continued by other students and researchers of the Machine Learning Group at the Department of Computer Science of the Universidad Autónoma de Madrid (UAM) with the objective of creating one of the first Python libraries related with FDA. Some of the reasons for the creation of the package are the increase and high utilization of Python in the areas of statistics and machine learning during the last years and growth of interest in FDA. This software is expected to hopefully be useful for many Python users worldwide, which will have multiple functional data tools available for their scientific projects.

1.1 Goals and scope

This project is focused on the development of visualization and analysis tools, expanding the work done by other students in this area, like Amanda Hernando [11] that developed new depth measures and clustering methods. The project goal is also to do a comprehensive redesign of the visualization tools. These have been grouped in a module and organized in a hierarchical structure. The interface has been homogenized, standardized and completed including interactive functionality to compare graphics and get insights of them. To do this I had to study among all the most important visualization packages to see which one fulfilled better the requirements our project was demanding.

The new graphs developed are the Outliergram, which is focused on detecting shape outliers, the DD-Plot, used to compare a dataset with different distributions and the Parametric Plot, which allows us to show two functions as coordinates. Moreover, to give more information when representing samples I added new features to the existing plotting method. These new functions give the library a wider range of functions to view data and plots.

One problem that the library was facing is the non homogeneity of the visualization methods. It may result difficult for the users to understand how to use all of them, so I decided to give them a common interface to make them all work in the same way. With the same purpose, it was studied the best ways to standardize the code imitating other scientific libraries, so the users are more comfortable with our software. With these changes the project succeeds in the goal of making the library easier for the user, something that is really important due to the complexity of the library and Functional Data Analysis as not necessarily the user should be an expert of it.

For a better understanding of the functions, an interactive module was added allowing the user to combine graphs and compare different representations of the same dataset in different plots. This provides users a much more comfortable and flexible experience.

One of the priorities of the project was also to develop high quality software, understandable for any user familiar with libraries in the Python scientific ecosystem. For this, besides the already mentioned, I pursued the objective of commenting and typing our code intensively. Online notebooks were created as examples (available at the website of the project [12]) of new functionality added to the project.

Finally I also took into account how well integrated was the code with different independent platforms as Jupyter, and how the new interactivity was functioning. Besides, the code is tested for Windows, Mac and Ubuntu, and compatible with Python 3.7 and 3.8 versions. Our main goal, was to make it work for every different Python user independently of the backend they use.

1.2 Document structure

The document is organized in the following chapters:

- Chapter 2, explores the state of art of the area, by providing a review of advanced tools for visual data analysis and studying the latest technologies and algorithms implemented. First, an analysis of the most advanced libraries for creating both static and interactive visualizations is done, revising the functionality they provide, their limitations and positive qualities. Afterwards, Functional Data Analysis and the package containing the new functionality developed in the project (scikit-fda) are explored, explaining its main modules and purposes. Finally, I describe the main visualization methods developed, their formulas and what information these graphics give us.
- Chapter 3, describes the software development process followed to develop the project. It describes all the steps made (analysis, design, implementation and testing) and the main decisions taken in each one.
- Chapter 4 shows the illustrations for all the functionality implemented during the project, from the visualization methods to the interactive graphics.
- Finally, chapter 5 ontains a summary of the contributions made in this project and some conclusions. Different possibilities for future developments are explored as well. Finally, I will provide my personal assessment of the project, including a review of the acquired skills that I have applied and the new ones that I have learned.

2

State of the art

This chapter explores some advanced static and interactive tools for visual data analysis, as well as the visualization functions developed in the project. First, a review of software packages for static and interactive visualization is made. Their functionality, advantages and disadvantages will be discussed. The study focuses on those aspects that are relevant to the current project. Special attention will be devoted to Matplotlib, which is one of the most common visualization packages in Python, and is used extensively in our project. The most salient aspects of its Application Programming Interface (API) are highlighted as it helps to understand some design decisions explained during the following chapter. Then, the scikit-fda library, its different modules and the functionality provided by each module are explained. Finally, an exploration of the visualization tools developed in the project is done, explaining the basic functional data notions that sustain the necessity for them, how they work or what can they do. These tools include the expansion of others already implemented, like the curves and surfaces visualization or new ones like the Outliergram, DDPlot, etc.

2.1 Visualization software

In this section, the most advanced visualization packages nowadays are explored, explaining their characteristics, how well they fit into our project and which one we will be using (Matplotlib).

2.1.1 Plotly

Plotly [13] is one of the most advanced open-source visualization libraries developed in Python. Despite this, it is also compatible with other multiple programming languages such as R or MATLAB. A basic version of this library is distributed under the under the MIT license (X11 license), which gives permission for both private and commercial use, modification. Furthermore, it is compatible with other licenses. Additional functionality intended for large companies, such as Chart Studio Enterprise or Dash Enterprise, is provided at a cost.

This toolkit is web-based, can be accessed by multiple GUIs like Jupyter and provides an extense variety of graphs. Plotly figures can be exported to a static file like an SVG, PNG, PDF or to HTML format that can be displayed in websites. This HTML format incorporates interactive visualization tools, such as hovering, zooming or showing and hiding elements thanks to the legend. It also has the possibility of adding widgets to interact with the figures.

The main disadvantage plotly presents is that it is not very common in other scientific libraries, having less dependants. This decreases the possibilities of maintenance due to there is no incentive to keep the library up to date. Furthermore, its use also hinders mantaining a similar structure to other SciKit packages that commonly use Matplotlib.

2.1.2 Ggplot2

Ggplot2 [14] is a library written in R that forms part of tidyverse, which is a collection of R packages created for data science. It is very popular due to the use of the concept of Grammar of Graphics [15]. This concept divides in layers the components of a graphic, simplifying the readability of the code generated thanks to creating graph by combining them.

Grammar of Graphics is a technique created by Leland Wilkinson with the aim of establishing a standard way of creating any type of graphic from any context, allowing us to divide the necessary components that create a graphic. As it can be seen in Figure 2.1, its different components form a pyramidal structure depending on its importance. The three fundamental pillars used to describe a graphic are the data to be plotted, the aesthetics defining what data is wanted to be displayed (for example the axis) and the geometric object used to plot our data (lines, points, bars, etc.). Without this three components it is not possible to create a graphic, but the ones above them are optional. The optional components are facets (used to create subplots), statistical transformations (percentiles, median, etc.), the coordinate system and theme used. The theme describes features not related with the data directly like legends or colors.

This layered framework for plot creation facilitates the specification and analysis of graphs. There is an implementation in Python that makes use of this framework (*plotnine*). In principle, it could have been used for this project. However, the usage of the library is rather limited, which rises the possibility of its support and maintenance being discontinued in the future. Besides, it is also important that other scientific libraries use it to have the most standard code, making it easier for the user.

One disadvantage of ggplot2 it is that it doesn't have its own interactive module and it is commonly combined with plotly (Section 2.1.1) to create interactive graphics. Other tools in R like ggiraph have appeared to provide also this functionality to ggplot2, giving an alternative to plotly.

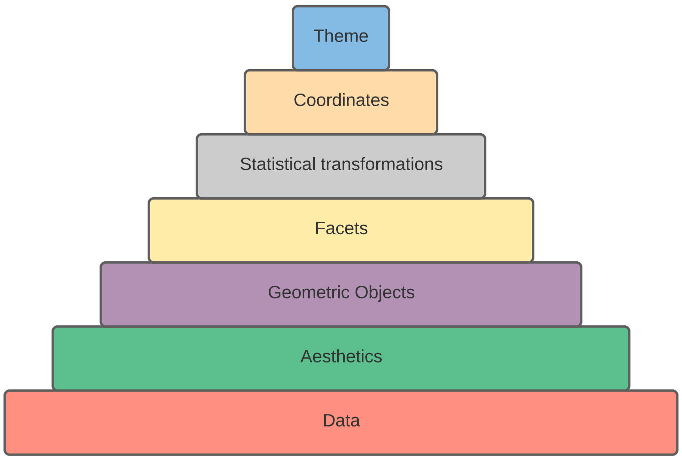


Figure 2.1: Main layers of the Grammar of Graphics

2.1.3 Matplotlib

Matplotlib [16] is one of one of the most widely used open-source packages for Python and its purpose is data visualization. It is known thanks to its extensive functionality and compatibility with other libraries like Numpy [17]. It allows the user to plot any type of data from curves (plot), points (scatter), surfaces (plot_surface), heatmaps, etc. It also has the possibility of creating 3D Axes, which in this project logically are very useful and allow to plot clearly not only curves but also surfaces or stem plots (useful to plot a discrete sequence of data, example in Figure 2.2). Another advantage of Matplotlib is that other packages in the Python scientific ecosystem like scikit-learn [18] are based on this library. Its wide use means that the library is likely to being maintained and updated in the short to medium term. Perhaps, other libraries have more advanced functionality but a priority of the project is the stability of the libraries used, so a popular one like Matplotlib provides us this. Besides, as in this project is pursued the standardization of our code, it is a great option to use similar conventions and API to other packages and to facilitate the use of scikit-fda's visualization tools by programmers who are already familiar with Matplotlib. Another interesting aspect, it that thanks to pyplot it gives the user a plotting framework similar to the one offered in MATLAB [19]. Due to this reasons, Matplotlib is the library used in our project.

Besides the different and multiple representing functionalities it offers almost a total control in style, visualization settings that can result useful for our project. Matplotlib also has extra toolkits like *mplot3d* or *axisartists*, and very useful third party packages like seaborn 2.1.4.

Matplotlib uses as its canvas an object of type Figure, which is a collection of different Axes or subplots in which the data is represented. The main advanced functionalities used to develop the project and the functionality that allows us to interact with graphs are:

- **Matplotlib events:** they are used to connect any element of our represented figure to a

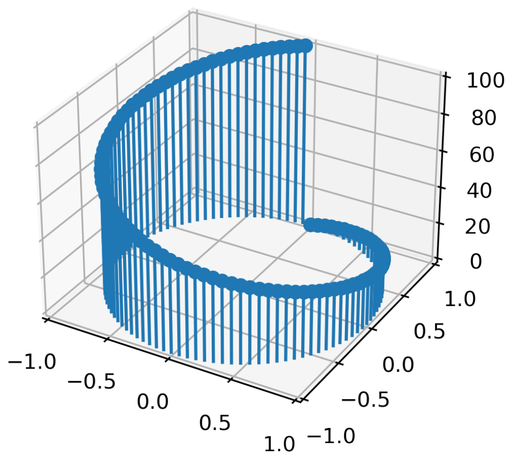


Figure 2.2: Example of an stem plot created with Matplotlib.

callback function. This events are triggered by a selected action such as the pointer's hovering an axis, selecting a point in a scatter plot, moving the mouse and others. For a specific action, the corresponding functionality is implemented in the callback function associated to it.

- **Artist class:** it is an abstract class for all the objects that are rendered (Axis, Figure, plots, scatterings, etc.). This is fundamental for the interactivity, as the callback functions can use its properties to edit the transparency of any sample (set_alpha), modify its color, plot new functions or add annotations. Depending on the representation function used, there are different types of artists like Line2D objects when plotting, PathCollection when scattering, Poly3DCollection when plotting a surface, etc. These different artists have different extra functionalities. For example, in the case of PathCollection they can be selected, which is used for the interactivity module. Other type of Artist that also exist are Patch objects, that are commonly shapes or Annotation objects (labels with text).
- **Matplotlib widgets:** they allow us to interact with graphics by triggering a Matplotlib event when a widget is used. They are represented inside an Axes object and have the advantage that work in every GUI backend. There are multiple types of widgets, like CheckButton, Slider and TextBox. It is possible to define callbacks for their activation in order to modify the Figure.

2.1.4 Seaborn

Seaborn [20] is a Python visualization package based in Matplotlib. The most clear advantage it has, is the upgrade on the previous graphics, obtaining better plots with the same code. Another feature that seaborn presents is the existence of some extra plots oriented to statistics, like regression plots or facet plots. This plots can be also created in Matplotlib, but require more code to be created. Besides, the variety of color palettes offered by seaborn is wider and present differents usages like categorizing, indicating importance with a sequence of colors or thanks to a diverging palette giving extra importance to the extreme point and under emphasizing the central. The main disadvantage of seaborn is that it is not used in other scikit projects and the advantages aren't enough to justify using this library.

Another library also based in Matplotlib is Yellowbrick. This package has been considered excessively complex, as it uses the *scikit-learn* API, which is intended for machine learning workflows, as the API for the plotting methods.

2.2 Functional Data Analysis

Functional Data Analysis (FDA) is a branch of statistics that deals that data consisting in curves, surfaces or in general, functions defined on a continuum. Due to new advances in the area, its use has grown considerably along the years. Despite this, because of the complexity of some datasets that comprehend multi-dimensional spaces, it is fundamental to create a good visualization module that allows the user to visualize data.

For the sake of completeness, and to provide the context of the contributions made, an overview of the characteristics of functional data is made. A sample of functional data consists of a collection of instances, such as curves, surfaces or general functions whose value depends on a continuous parameter. These instances can have different dimensions depending on its codomain. For example, it is not the same a curve in a 2D graphic than a curve in a 3D graphic.

Most of the software packages for functional data analysis are written in R. They can be retrieved from the Comprehensive R Archive Network (CRAN) [9]. One of most comprehensive FDA packages is fda [7], developed by one of the authors of Functional Data Analysis [4] (J. Ramsay). It was designed to obtain a tool containing the concepts and functionality explained in the book. Such as fda, fda.usc [6] [21] is another package with extense functionality. Other tools, such as the rainbow package [22] [23] are smaller as they contain more specific methods. This package was created to visualize functional time series and identify functional outliers. Outliers are instances of our sample that represent an atypical case in it, showing atypical values (magnitude outliers) or shape (shape outliers). One of the methods the rainbow package has to detect and view these outliers is the BoxPlot, which also exists in our library, scikit-fda. These outliers can be useful in the visualization methods and multiple visualization tools use them to give more information to the user.

A package that cannot be found in CRAN is tidyfun [24], that forms part of a collection of R packages for data science called *tidyverse*. Tidyfun allows to manipulate data with other tools from the tidyverse. Its purpose is giving the users an easy experience with functional data analysis and data wrangling thanks to what they call tf vectors. Functional Data is stored as tf vectors that can be visualized or manipulated. It contains some advanced visualization methods such as the lasagna plot. The lasagna plot is a heatmap containing the different observations in rows and uses colors to indicate the values of them.

2.3 scikit-fda

The software project oriented to FDA in which I have collaborated is scikit-fda [1]. Its purpose is to offer a generic and wide range of tools to the user, so the library can be applied to any area related with FDA.

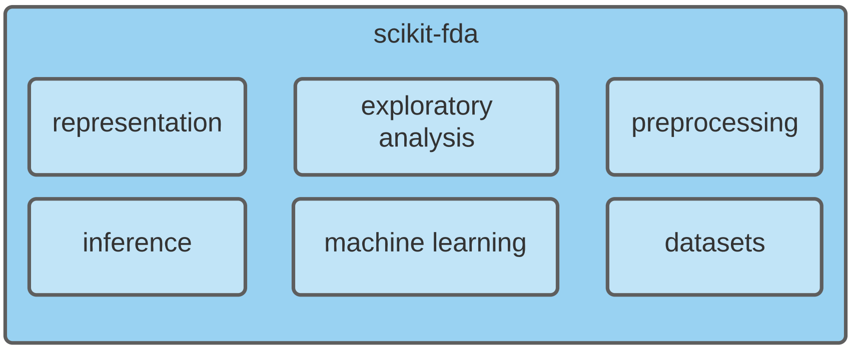


Figure 2.3: Main sections of scikit-fda package

Figure 2.3 is a scheme containing the six main modules of the package:

- **Representation:** This module provides support for the representation and storage of functional data. There are different ways of storing a functional datum. One option is to represent the funtion as a one-dimensional array of values at a set of discrete observation points in a grid (FDataGrid object). Another possibility is to use a basis representation. Once the basis is fixed, a functional datum is represented by the coefficients of a expansion in that basis (FDataBasis object). Both of them have a parent abstract class, FData, that has its common functionality.
- **Exploratory analysis:** this module includes the tools (classes, methods) necessary to characterize, analyze and visualize the data. It is composed of different submodules. Of particular importance for this project is the visualization submodule. This submodule contains methods to visualize clusters, Functional Principal Components Analysis (FPCA), etc. Finally, the methods for computing the depth of a functional datum within a sample are included in the depth submodule. These depth measures are extensively used for data visualization and outlier detection. The tools for the estimation of summary statistics (contains methods such as mean or median) and outlier detection are grouped in the submodules named after them.
- **Preprocessing:** the classes and methods contained in this module are used to process and prepare the data for subsequent analysis. Some of its functionalities are smoothing that

reduces noise, registration that aligns data to eliminating phase variation and dimensionality reduction to simplify the data.

- **Inference:** This module provides tools for hypothesis testing and the estimation of population properties from data samples of finite size. Some of its main submodules are ANOVA and Hotelling.
- **Machine learning:** set of tools to automatically infer predictors from data. It contains three submodules: classification, regression and clustering. The classes and methods included in this module follow the design patterns of scikit-learn and provide interfaces compatible with this package. This ensures that scikit-learn's tools can be used in combination with predictors for functional data (e.g. model evaluation and selection tools, cross-validation grid-search for hyperparameter estimation, and others).
- **Datasets:** contains all the functionality used to obtain some datasets from the CRAN and UCR repositories. This provides the user an extense variety of datasets to work with and the possibility to add even more.

2.4 Visualization and tools for functional data analysis

In this section, the depth measures used in the visualization tools are described. Then, a description of the graphics that have been implemented, the information that can be extracted from them, together with guidelines for their use are provided.

2.4.1 Depth measures

Consider a sample consisting of a collection of curves. A depth is a measure of how central a curve is with respect to the curves in the sample. Different definitions of a centrality measure can be formulated. In this section, some important depth measures are presented.

2.4.1.1 Integrated Depth

An integrated depth consists in the integral of a multivariate depth along the continuous parameter on which the functional data depend. An instance of these types of depth functions is the one introduced by Fraiman and Muniz [25] with the next equation:

$$D(x) = 1 - \frac{1}{2} - F(x)$$

This depth measure is used in several examples in Chapter 4.

2.4.1.2 Band Depth

The Band Depth [26] is a depth measure based on the concept of bands. A band is the area delimited by two curves in the sample. The Band Depth of a curve of our sample is the fraction of bands obtained by two instances that enclose completely the curve whose depth is being computed.

2.4.1.3 Modified Band Depth

The Modified Band Depth (MBD) as well as the Band Depth [26], relies in the concept of bands. MBD To compute the value of the MBD, one measures the fraction of time each curve is enclosed for each of the bands, defined by a pair of curves in the sample. This quantity is averaged for all possible bands in the dataset.

Let $x_1, \ldots, x_n$ the different functions in our dataset. The measure of centrality estimated in terms of the MBD is:

$$MBD_{\{x_1,\ldots,x_n\}}(x) = \binom{n}{2}^{-1} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{\lambda(\{t \in \mathcal{I} \mid \min(x_i(t), x_j(t)) \leq x(t) \leq \max(x_i(t), x_j(t))\})}{\lambda(\mathcal{I})},$$

In this expression one considers the band formed by each pair of curves x_i and x_j is defined in $\mathcal{I} \times \mathbb{R}$. The denominator ($\lambda(\mathcal{I})$) is the length of $\mathcal{I}$, the interval within which the functions in the sample are defined. The numerator is the fraction of time in the interval $\mathcal{I}$ that $x(t)$ spends within the band defined by $x_i(t)$ and $x_j(t)$.

2.4.1.4 Modified Epigraph Index

The Modified Epigraph Index (MEI) is not exactly a depth measure, as it doesn't represent centrality, but is explained in this section due to its high relationship with the Modified Band Depth. MEI is the average time a sample of our data stays under the rest of curves or surfaces of our dataset. It is measure related, albeit simpler, to the MBD in which the band is replaced for the area below a given sample.

Let $x_1, \ldots, x_n$ be the collection of functions in our sample. The expresion of the MEI is:

$$MEI_{\{x_1,\ldots,x_n\}}(x) = \frac{1}{n} \sum_{i=1}^{n} \frac{\lambda(\{t \in \mathcal{I} \mid x_i(t) \geq x(t)\})}{\lambda(\mathcal{I})},$$

In this case the numerator for the ith term in the summation is the fraction of time that the function $x(t)$ is below $x_i(t)$. A naive implementation requires carrying out three nested loops. The time complexity of this algorithm is $O(n^2m)$ (being m the number of instances of the sample and n the number of points that conform a curve), which is rather poor. It is possible to reverse the order of the sum and the integral The computation of this quantity can be vectorized by reordering the curves at every point in the domain with the *rankdata* Scipy function. This avoid one loop and taking into account the worst case the complexity drops to $O(n * \log(n)m)$. This process improves the efficiency of the algorithm, which makes it possible to carry out the computation of this index in large datasets faster.

2.4.2 DD Plot

The DD Plot or Depth vs. Depth Plot [27] [28] [29] is a tool to compare two samples. The comparison is done by computing the depths of the instances of a dataset in the two mentioned samples. This gives us a pair of depths for each instance of the dataset. Every pair is formed by the depth of an instance obtained in both samples.

The graphic consist in plotting this pairs of depths as coordinates to see the relationship between both, as the higher values obtained, the more related they are. In the same graph, a straight line of slope one is plotted for reference. If both samples have the same distribution of depths, the points in the plot are located along this line.

This line divides our graph in two spaces, the points lying below that have a function more similar to the first distribution, whereas the one above it that have a closer form to the second distribution. If both distributions are similar, the pair of calculated depths for every instance should be close in value, tending to form a straight line very similar to the auxiliary one.

This type of plot can be used also for classification. Given a test function, the goal is to determine whether this instance belongs to the first or to the second sample. To this end the depths of this test function with respect to each of the samples are computed. Then, the point whose coordinates are these depth is plotted in the graph. If this point lies below the reference line, it is assigned to the first distribution. In the opposite case, when it lies above the reference line, it is assigned to the second sample. This is the DD-classifier [30], currently being developed by Pedro Martín as a part of his undergraduate thesis.

2.4.3 Parametric Plot

The Parametric Plot [4] is he graph defined by a set of points in the real plane whose coordinates are given by the values of two functions. For example, consider the following functions:

$$f(x) = x^2$$
$$f'(x) = 2x$$

They both depend on the same parameters and have the same representation (x, f(x)) and (x, f’(x)). The parametric plot instead represents (f(x), f’(x)), in order to see for example how they change with respect to the other. In Figure 2.4 the parameterization can be observed.

This is very useful for differential equations as understanding the evolution of both simultaneously

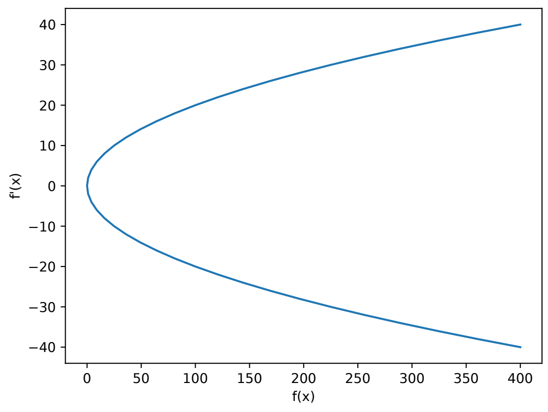


Figure 2.4: Parametric plot of f(x) and f'(x)

gives us important insights. It can be also used to compare an instance of our sample with its average to find differences between them. For example, in Functional Data Analysis [4] there is an example where they use this plot to get insight in the way a child walks. To do this they compare two functions, one representing the hip angle and the other one representing the knee angle. In the results chapter, this simulation is done with the tool (Section 4.3). Plotting the average function versus the case of only one child can show problems of the way of walking which can be difficult to realise otherwise.

Despite the software part is covered later, there are two different cases of entry in our function:

- A function of type $f : \mathbb{R} \longrightarrow \mathbb{R}^2$ that has a codomain of dimension 2 and domain of dimension 1. This would be directly processed by our function.
- Each of the two functions of type $f : \mathbb{R} \longrightarrow \mathbb{R}$, has a codomain of dimension 1 and domain of dimension 1. If they are joined as coordinates, the result obtained is a function of the first type that can be normally processed.

2.4.4 Outliergram

The Outliergram is a graphical tool for the identification of shape outliers for functional data [31]. To understand how it works, two important concepts need to be introduced: magnitude and shape outliers. Magnitude outliers functions in a dataset whose values are atipically low or high with respect to the other functions in the sample. They can be found using measures of centrality such as depth methods. Shape outliers aren't as easy to characterize as magnitude outliers. In particular, depth measures cannot be used to detect them. This is because what characterizes them is the anomalous shape or form they present. With the Outliegram, a new option is given to solve this issue and find this last kind of outliers. Removing these kinds of atypical instances can help us preprocess our data in order to make sample estimations that are more accurate.

The scikit-fda package includes a number of graphical tools that can be used for outlier detection. For example the Magnitude-Shape Plot [11]. This method uses a depth method to compute the directional outlyingness of each instance, indicating the centrality of it. Thanks to the directional outlyingness it is possible to detect the outliers. The difference between the Magnitude-Shape Plot and the Outliergram, is that with the Magnitude-Shape Plot is possible to distinguish magnitude and shape outliers thanks to the directional outlyingness method. By contrast, the outliergram is especially useful to identify shape outliers in the sample. The authors suggest to use the outliergram in combination with the functional boxplot [32] (already implemented in the library) to get also the magnitude outliers.

This visualization method utilized two depth methods: Modified Band Depth [33] [11] and Modified Epigraph Index [34]. The first one, was already implemented in the library, the second one, has been developed in this project.

2.4.4.1 Relationship between MBD and MEI

The interest of our plot relies on the interaction of these two depth functions, and that is why the graph consists on plotting MBD (Y-axis) against the MEI (X-axis). When both of them are scattered in a graph, almost all of the points tend to form a parabola. This is because the fact MEI has very low or high values means that our sample is located at the lower or upper part of the graph, respectively. If the instance has extreme values, the MBD should be low as it is contained by bands for a low proportion of time. The points that correspond to a high MBD value indicate this instances are centered, having then approximately the same number of curves above as below it (central MEI value).

The way to detect these shape outliers is to spot points that the lie far down from our parabola. If a point has an average value (around 0.5) in its MEI, this means the curve values, are very central as it stays the same amount of time under curves and above them. As the datum or instance has a central MEI value, it would be normal to expect that also the MBD is high and as it is a central value, it should be contained by many bands. Due to the different shape a curve can exhibit, as appears in Figure 2.5, it won't be contained by as many curves as expected, then we can conclude it has a different shape. Besides this, even if it is not specifically designed for it, the points of the corners of the plot could represent some magnitude outliers, as they have lots of curves behind or over them most of the time.

2.4.4.2 Shape outlier detection

Everything that has been talked about until now are non-computable ways to distinguish these shape outliers. Detecting an outlier by seeing that its corresponding point lies far from the parabola it is not a bad method but can be improved. To get which points are indeed outliers, the first step is to compute the parabola that this points define. Taking into account our sample $x_1, \ldots, x_n$ and its corresponding MBD($mb_i = MBD_{x_1,\ldots,x_n}(x_i)$) and MEI($me_i = MEI_{x_1,\ldots,x_n}(x_i)$).

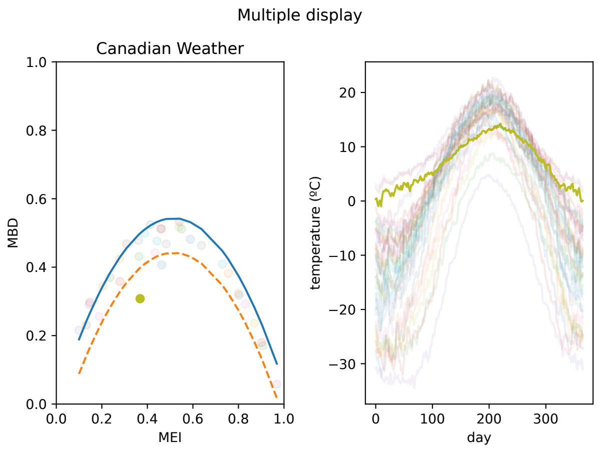


Figure 2.5: Outliergram example of a curve that exhibits different shape

$$p_i = a_0 + a_1 me_i + n^2 a_2 me_i^2,$$

$a_1 = 2(n+1)/(n-1)$, $a_0 = a_2 = -2/n(n-1)$ being n the number of functions of our sample.

With the array of points $p_1, \dots, p_n$ is possible to plot the parabola. The vertical distance of each point to the parabola can be computed by subtracting to the last array its corresponding mb_i, as represented in the following function:

$$d_i = p_i - mb_i,$$

A common way to determine outliers and the one used to compute the dashed parabola that separates the outliers is using the interquartile range, which was already implemented in our library. The detection algorithm determines that a point is an outlier if its distance is bigger or equal than the sum of the third quartile and the inter-quartile range of d. It follows the next inequation:

$$d_i \geq Q_{d3} + 1.5\mathrm{IQR}_d$$

For a clearer vision to the viewer, a copy of the previous parabola is shifted down to get the curve in our graph that defines the limit between outlier and non-outliers.

Even if it is a very good method to keep track of this kind of atypical curves, it should be taken into consideration that if a function presents a very low or high MEI (near 0 or 1), it must present a low MBD so it is not detected as an outlier. Some option to solve this, would be shifting this kind of cases to more centered values before using the Outliergram.

3 Software development

During this chapter, I explore the different steps taken in the Software Development Process and the main decisions made to develop the project. The initial phase is planning, which in my project consisted on estimating the amount of time needed, the organization followed during the course and choosing a software development process model. I decided to opt for an incremental model, as my job was divided into two big parts, developing the new tools and after that, standardizing it with the previous methods to add interactivity. With this strategy, I could make the project by small steps, which eased my work during the year.

As *scikit-fda* is a project in which many people work during the year, the collaborators decided to organize weekly meetings every Monday in order to share our advances, new knowledge or ideas for future parts of the project. These online gatherings came in handy, as everybody could learn from each other work and keep up with the latest updates in the library. During this chapter the tools used during the development have been omitted and can be found in Appendix A.

3.1 Analysis

In this stage I developed a software requirements specification, defining functional and non-functional requirements, in order to specify what I wanted the project to do and under which conditions it does it, respectively. The functional requirements obtained related with specific new functionalities are:

- **FR(1)** The visualization module should have a new graphic tool that represents the Outlier-gram, enabling the user to detect shape outliers by following the equations defined in Section 2.4.4.
- **FR(2)** The visualization module should have a new graphic tool that represents the DDPlot, enabling the user to compare a sample to different distributions thanks to a depth measure, as defined in 2.4.2.
- **FR(3)** The visualization module should have a new graphic tool that enables the user to plot parameterized curves Section, as defined in 2.4.3.

- **FR(4)** The visualization module should extend the representation of curves by adding the possibility of using a gradient of colors in the different samples.

The functional requirements that are thought for all the module are:

- **FR(5)** The visualization module should have a common interactive module that allows users to interact with graphics and combine them in different subplots in the same figure.
- **FR(6)** The visualization module should allow users to hover points and get an more information about the sample hovered thanks to a temporal floating annotation.
- **FR(7)** The visualization module should allow users to click points and highlight the sample clicked in all the different subplots present in the current figure.
- **FR(8)** The visualization module should allow users to add widgets attached to a criteria (for example a depth measure) that allows the users to interact with the different subplots of the figure highlighting the curves that fulfill the condition the criteria has.

Some of the non-functional requirements are common for all the library and had been applied in other earlier projects, and are the following:

- **NFR(1)** The software designed should be cross platform, as it has to work in the main operating systems: Windows, Mac and Linux.
- **NFR(2)** The software should have a complete documentation including explanation of arguments and examples of the use of the functionality.
- **NFR(3)** The software developed should be compatible other scientific libraries like scikit-learn [18], Matplotlib [16], NumPy [17] or SciPy [2], in order to help the user and make them comfortable.
- **NFR(4)** The software must be implemented in Python.
- **NFR(5)** The software is public and open-source.
- **NFR(6)** The software should follow the PEP 8 [35] and PEP 257 [36] standards for coding and documentation.
- **NFR(7)** The project should be stored in GitHub due to its version control, automatic testing for all the platforms, style checking or code coverage.
- **NFR(8)** The software should include unit tests.
- **NFR(9)** The software should be compatible with Python 3.7 and 3.8 versions.
- **NFR(10)** The software should be usable in every different Python environment: Jupyter Notebook, IPython, QTConsole and others. This is specially important for the interactivity module and its possible incompatibilities.
- **NFR(11)** The software code and comments should be written in English.

- **NFR(12)** The software code and comments should have the correct style and follow the rules determined by the Wemake-Python-Styleguide (WPS) specified in out setup.cfg file., enforcing other already mentioned like PEP8 thanks to flake8.

Furthermore, for a better undertanding of the requirements of the module I made a use-case analysis (Figure 3.1), that also helps to understand the actors and possible processes that can happen:

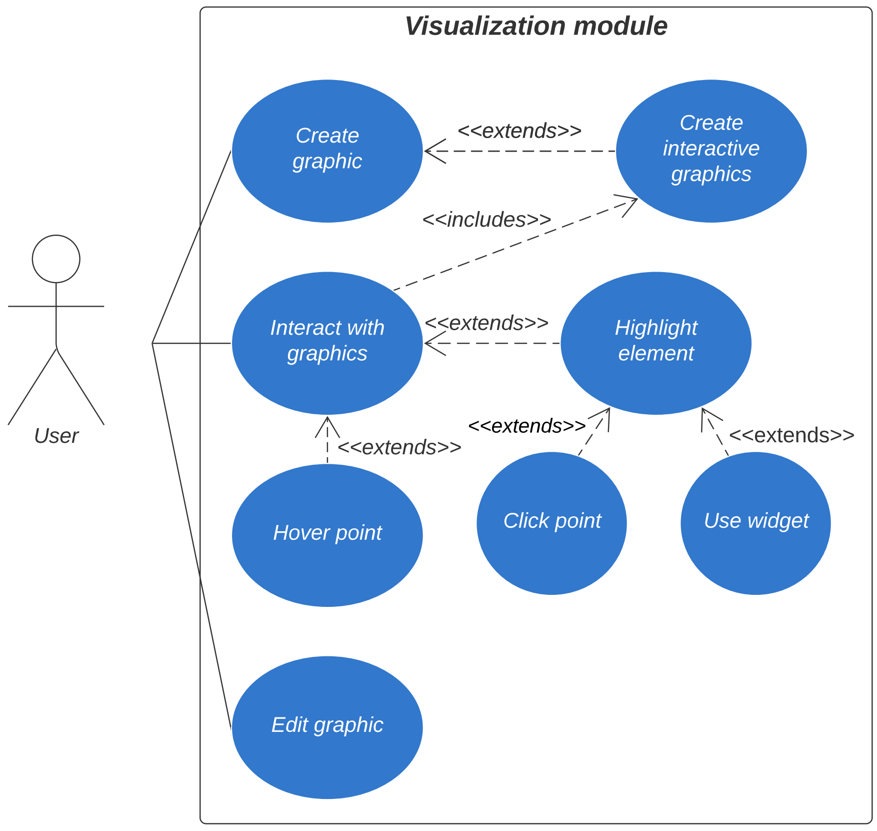


Figure 3.1: Use case diagram of the visualization module.

After the software requirements specification, I developed a Gantt Chart to organize the estimated time for each of the functional requirements and chores of the project. This diagram is available in Appendix C.

During the analysis I had to decide which visualization library I was going to use for the project and as explained in 2.1, I opted for Matplotlib.

Another important aspect I had to take into account in step of the software development process was to understand the existing project and its main modules (Figure 2.3 [1]), which have been explained in the previous chapter.

Furthermore, another important aspect of the project is the inclusion of it into the SciPy Toolkits

(SciKits). This collection of toolkits are scientific opens-source packages united under the SciKit brand. The packages in it should be under OSI-approved licenses (Section 3.6). My project is focused in the development of the visualization submodule of exploratory analysis, that is in charge of the tools that create the graphics that represent our functional data plots.

3.2 Design

During the design, I went through two periods: a first one where I had to create the new algorithms and functions of the new tools required and a second one where I had to think of a way to make all the visualization modules alike. By doing this, they have a simple interface that is similar to all of them. Finally, I had to create a class that can join every module into a multiple graph where the user can interact with the graphics.

After all the design, the module is totally integrated and has the following class diagram (Figure 3.2).

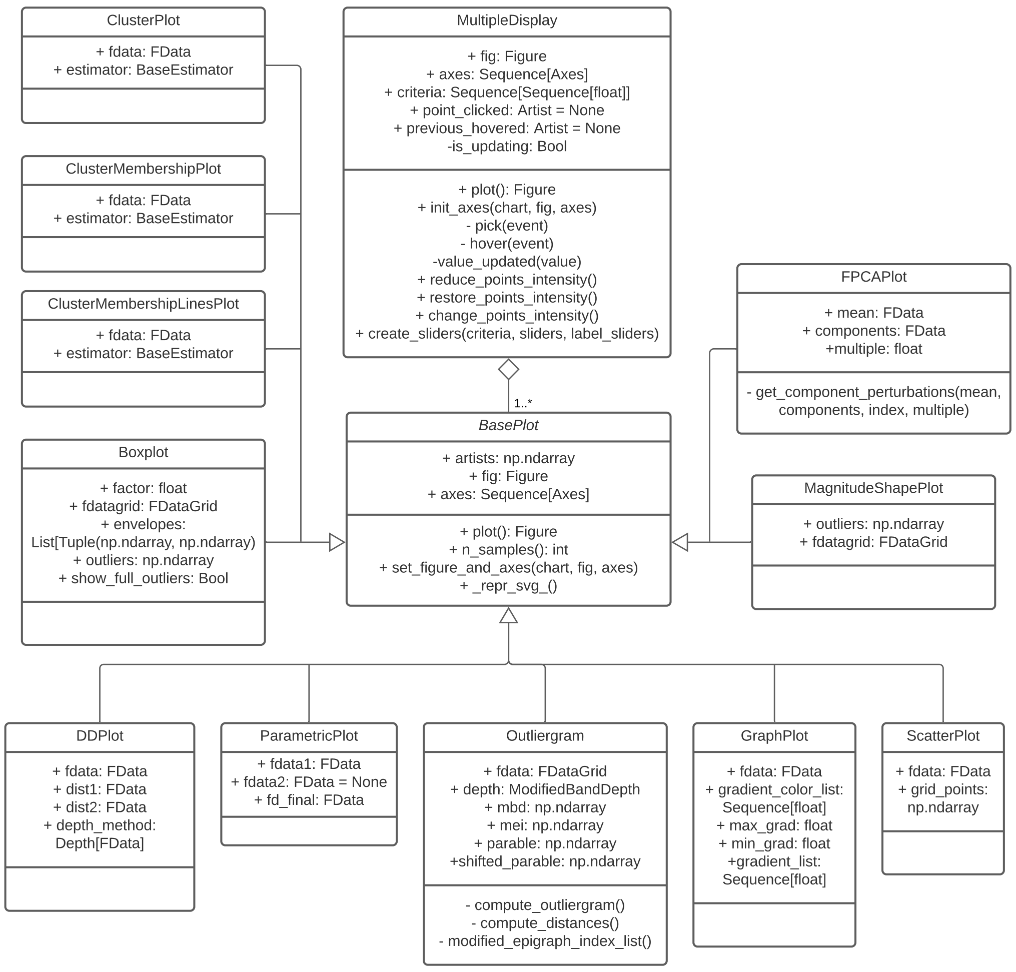


Figure 3.2: Class Diagram of the visualization module of the project

All this classes can be found in the directory skfda/exploratory/visualization folder of the project, which can be consulted in GitHub.

In the next subsections, there is a description of the main design decisions taken with the classes. During this explanations there are notions and concepts explained in section 2.3.

3.2.1 BasePlot Class

As it can be seen in the class diagram, BasePlot is an abstract class of which all the plottable classes inherit, giving a common structure for this kind of objects. This allows scalability for this module as if new visualization methods are added following this structure they would be also available for a multiple and interactive plot.

My main concern when designing the BasePlot was the necessity for a simple interface that all the classes could follow, without exceptions. The problem encountered initially when creating this class was that every function in the different files had been implemented by different people (different projects), following different styles so I had to find something in common to all of them. The answer is that Matplotlib uses a Figure object that represents the plot and that is formed by a sequence of Axes corresponding to the different graphs a plot can have. As in every visualization function this figure and axes had to be initialized manually by some parameters or automatically if none where given, an abstract method was created to initialize them.

Despite I already had where to plot our data, I realised that not every visualization function plotted FData objects, actually some of them plotted some computations done with the FData, for example, depths in DDPlot or more than one FData could be used (Parametric Plot). So I realised the only thing the modules had in common was that they all represented in some ways instances of a functional data set or a combination of functional data sets. This was very useful to create our array of artists, a group of references to the Artist objects (graphical elements in the figure) obtained when scattering or plotting each of the elements in our data set and that is used in the interactivity part at MultipleDisplay, as the array contains information and can edit the different data plotted.

3.2.2 Outliergram Class

This module is the one containing the most complex algorithms developed in the project. As it has been seen previously this module also inherits from the BasePlot class and has a very simple input. The Outliergram is only compatible with FDataGrid objects and it computes the Modified Band Depth and Modified Epigraph Index of the selected FDataGrid.

Additionally, the parabolas are computed thanks to the formulas provided in the state of the art chapter, so the outliers can be checked visually. Despite the complexity of its algorithms, when encapsulating its functionality it has a very simple code with just the steps mentioned before. As this method does scattering, the references to the Artist are PathColletion objects.

3.2.3 DDPlot Class

This module has the advantage that allows FDataGrid and FDataBasis. It receives three functional data sets, one is our data and the other two represent the distributions that that are compared with the first. It is possible to select any depth method to compare our dataset to the distributions, which is done by fitting first our data with the depth, and after that comparing with the depth method every instance of our data with the two distributions.

The representation of both depths is also done in a loop as the Outliergram to get in artists all the PathCollection references to modify the intensity of the points afterwards in an interactive mode.

3.2.4 ParametricPlot Class

This module allows all types of FData and depending on the size of its domain and codomain, it receives one or two functional data sets. If the dimension of the codomain of our function is two, the method will be able to parameterize this dataset alone. On the other hand, if the codomain is one another function of the same type to do the plot will be needed. This two functions are be joined as parameters to obtain a function of the first type. In this case, as the method plots curves, the artists stores Line2D objects, the kind of Artist Matplotlib has for curves.

3.2.5 GraphPlot Class

This module was in charge of plotting any kind of FData, from curves to surfaces. My job with this module was to expand its functionality and allow representing datasets with a degree of colors depending on a parameter. This parameter is a sequence of floats that determine the color in the display of each of the curves or surfaces in our graph, like the result obtained when computing a functional depth to our data. To use properly this sequence and get the most representative gradient of colors, the sequence should be normalized. To do this normalization, there are optional parameters to set manually minimal and maximum values for the sequence. Usually, this sequence indicating the gradient of colors is the result of the computation of a depth, so in a quick glimpse the user can see the data selected and how it is distributed.

This list is used to compute the color thanks to a colormap which could be introduced by the user. With the typical colormaps, the instances with the higher values are represented with a more intense color while the ones with the lower value with a softer one. I defined as the default colormap the Autumn one, that can be seen in Figure 3.3. This was done due to it was the one that presented the most standard and gradual colors, from a soft one to a more intense one, being very flexible for every situation. Despite this, it can be selected in the inputs of our class initialization.

Figure 3.3: Autumn colormap.

As I mentioned before, depending on the dimension of the domain this module will plot curves or surfaces. Thanks to the design of the interactivity module the surfaces are also interactive due to their object obtained while plotting also inherits from the Artist class, so it has the necessary methods to work.

3.2.6 Rest of plotting classes

Besides the functionality already mentioned, the rest of the classes had to be changed too. The ScatterPlot had to be converted into a class but wasn't any trouble due to its similarity with the GraphPlot. Other visualization methods like Clustering had to be divided in to three different classes, depending on the different plot they represented (ClusterPlot, ClusterMembershipLinesPlot and ClusterMembershipPlot). The rest of the modules, FPCAPlot, MagnitudeShapePlot and BoxPlot, received the necessary modification to be transformed into a class and have interactive functionality.

3.2.7 MultipleDisplay class

This class is the responsible of the interactivity module and the relationship among the different BasePlot objects. It is formed by a sequence of BasePlot objects, each one with its corresponding axes. The most elementary functionality it presents is offering the possibility of plotting multiple graphics together that have the same number of samples represented, typically the same data in different ways. I realised this was extremely useful and decided to search for a way to highlight an element in all the graphs among the rest. After some time thinking, I realised that instead of highlighting the selected element I could make more transparent the ones not selected, in this way the whole dataset could be seen but specially the selected one.

One of the problems of having a big functional dataset is the difficulty of recognising the different instances represented in descriptive graphs that show outliers like an Outliergram or a MagnitudeShapePlot. Due to this, I decided to add an interesting interactive functionality which allowed the user to get more information of an instance of the sample if its being hovered (passing the cursor over the point that represents it). To do this I used the hover functionality of Matplotlib ("motion_notify_event") which activates a callback while moving the mouse over an Axes object. I created different annotations for each Axes instance which updates its location and information in case of hovering. The annotation contains the instance number and coordinates of the represented point. The instance number could be very useful for them to revise manually the data on the different graphs with that id and the coordinates give the position of the point which could represent depths or other measures.

Moreover, I thought of another idea of selecting elements different to picking which consisted on the use of widgets. I started researching for the best libraries of widgets python had until I found the two that best fitted into the project, ipywidgets [37] and the widget module Matplotlib incorporated. The first mentioned is the library Jupyter Notebook incorporates for interactive HTML widgets, including a wide variety and a very good graphical appearance. The widget section Matplotlib has, contains a smaller collection of widget types and doesn't have as modern aspect as its competitor. Due to this I initially designed all the project with ipywidgets. During the last month of the project, I realised that many users around me didn't necessarily use Jupyter Notebook, what was fatal for the first library and would incur on not respecting the **NFR(9)**. presented in the previous section. Finally I changed my implementation to Matplotlib for a total flexibility of the project, allowing all the users to enjoy the advantages of widgets. Despite this, a small sacrifice was made in the graphical appearance and usability.

With this change of widget implementation, I decided to focus our selection of widgets into Sliders, because from the catalog was the one that fitted most our purpose. The sliders are used to order the data instances thanks to a criteria created by the user. The criteria is used to order our data, using for example the centrality functions I have already mentioned, depths. If a point has low depth, it will be located in the lowest points of the slider and viceversa.

One of the advantages of the implementation, is the possibility of adding as many widgets as pleased. Because of this, all the widgets have to be updated every time one widget or point is clicked. This is important as sliders represent notions like depths which should always have coherent values. As widgets and the clicking points functionality does the same changes in variables in different ways, it was important to add the functionality that made them compatible.

3.3 Implementation

As scikit-fda is a collaborative project in which many people have participated during the years, one of the top priorities is to improve the readability and documentation. Besides this, as it is an open-source project that can be used by anybody and can receive the help and collaboration of other developers, it is important not to only think about easing the job for people that want to use our tool but also for people that will be making modifications in the future.

Before I started the project, two of the main Python Enhancement Proposal (PEP) were already being followed PEP 8 [35] and PEP 257 [36]. The first of them, PEP 8 (Style Guide for Python Code) is a code convention indicating rules for the coding style of the project. The second one, PEP 257 (Docstring Conventions) is in charge of the rules related with how the project should be documented.

In the Appendix A, I have explained the tools used to ensure this styleguides are followed (wemake-python-styleguide) and generating the best possible documentation from the docstrings (Sphinx).

3.4 Testing

Testing is a vital part of every software project, and this one is not an exception. It is very important as it is the best way to detect regressions, bugs that may have appeared while adding new functionality or due to changing older modules. Furthermore, new improvements could be detected to improve the software developed. As it was a project already in development, some of the previous tests had to be modified with the new design and some new visualization features had to be tested. During the project I developed two types of tests: unitary test and user tests.

As my job with the visualization methods was to create graphics, it was not very easy to test all the functionality I developed, so what I did was to create tests for the data that needed to be computed to be plotted. Nevertheless, as some data that was going to be plotted were depth measures that had its functionality already tested, it was not tested again as this tests would not give us extra information. I studied some ways to do image comparison tests with Matplotlib in order to see if the graphs were correct, but I decided finally not to use them because they where hard to develop for the advantages it actually gave. Finally, the new unitary tests were focused on the outliergram and the computation of the MEI. In Appendix A.2 the tools used for doing unitary tests are explained.

Besides the unitary testing, I also developed a usability test with the objective of making the app easier for typical users, finding bugs or getting a different perspective of the software implemented. The technique used is Thinking Aloud [38], considered by Nielsen as one of the best methods to test the usability. It consists on a test where the user is asked to verbalize what he is feeling while using our interface and the questions he has and the doubts that may appear. It is a good technique that allowed us to improve our interface and which was partially implemented in our weekly meetings when I showed the progress online to the rest of the people in the group and they gave me their opinions. This feedback allowed me to see errors that I didn't realise of or new ideas that could improve the user experience. At the end of the project a total review of the tool was done with one of the main developers of the library, Carlos Ramos, who hasn't tried yet himself the new module. The usability test results can be consulted in Appendix B.

After this stage, the last step in software development is the maintenance. It basically consists on resolving bugs that may appear and are noticed by other developers or users. To do this, new issues can be created in GitHub to inform the developers of errors or new features to be implemented. Moreover, one of the objectives of following good coding styles and documenting our code properly, is that this allows to reduce the time spent on maintenance, as the code is clearer for other developers and it prevents bugs.

3.5 Integration

Another important aspect taken into account during the software development process is the version control of the project and how it was integrated. As mentioned in the non-functional requirements, this project used GitHub to upload the code and compare the new versions. More information abount Github can be found in Appendix A.4 with the rest of the tools used. The version control software it uses is Git, that allows the different developers to create and combine their own branches of the project with their new work. The use of branches, helps to avoid losing data thanks to having different versions of the project what helps to have a good quality software. To organize our branches, it was needed to select a model, in my case the project used GitFlow as it allows the different developers to work in parallel and as a consequence, faster and more comfortable.

The main branches conforming our model are develop and master. In the master branch, everything is up to the official last version of the project, whereas the develop branch receives the new functionality that is being continuously integrated. From time to time, a new version is created and the branch version is upgraded with the functionality that has been added to the develop branch (through a release branch). Usually these new versions are done when the new functionalities implemented in develop are complete. Other types of branches are feature branches, that are in charge of adding new functionality to the project, and are supposed to merge to the develop branch. In addition, there are also hotfix branches, that are used to solve bugs in master; and release branches, that merge the develop branch with the master in order to ensure that when a new version of the project is released, all of it is ready to be uploaded and has no bugs.

3.6 Licenses

One important concept in open-source projects is the choice of licenses. Besides the license your project has, it is important to revise the licenses present on its dependencies. A software license is a contract that controls the use and distribution of software and its copyright. As this is an open source-project the license used is BSD 3, that allows the commercial use of the library in projects, its modification, distribution or private use, with the condition of reproducing the copyright notice already mentioned. Some of the limitations it has are that it doesn't provide neither liability or warranty. BSD 3 appears as a modification of BSD 2 license which is very similar, but in the case of BSD 3, there is a third clause that doesn't allow to use the name of the contributors or the project with the aim of promoting any product without consent.

The relevance of licenses is not only important to understand due to the copyright of our own library but for the libraries used in our project. The packages should be used respecting their licenses and making sure they are compatible with ours. While selecting and investigating the visualization packages this was something I had to take into account due to its legal implications.

4

Results

During the chapter, I will explore the final results obtained thanks to multiple experiments of the new project's functionality. My goal is to explain the different use cases of the tools, how they work and what can be done with them. I will be also be viewing how they interact between them thanks to the interactivity module and its widgets. To show the changes and interactivity in a plain file I will try to show the elementary actions that can be done, like clicking, hovering or using widgets. More examples are available at the Appendix D and also in the website [12] examples section.

4.1 Outliergram

The data sample mainly used for the results is the one representing the data of the temperatures in Canada in different stations along the year, from the Ramsay's FDA package [39] available in CRAN. The sample represents the temperatures of different climates found in the country as can be seen in the following picture (Figure 4.1), using each color to represent the different climates:

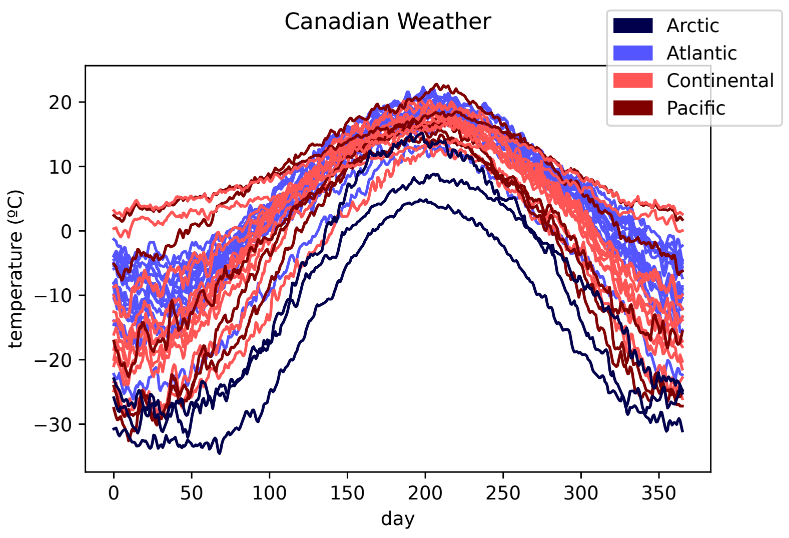
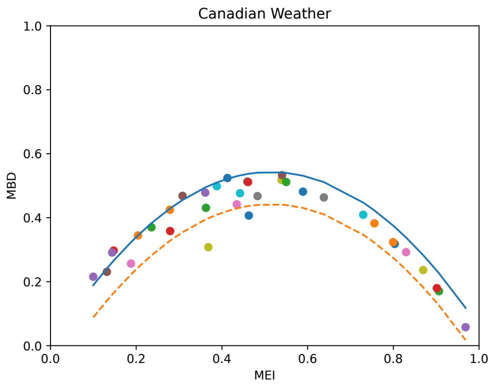
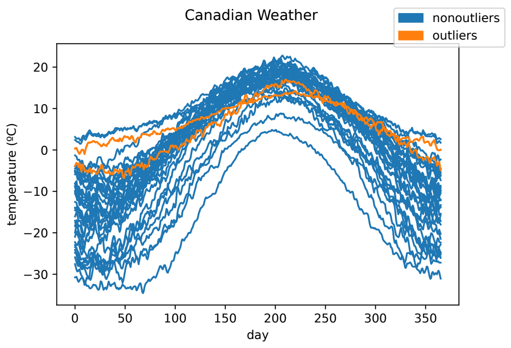


Figure 4.1: Representation of the temperatures during a year in the different stations of the Canadian Weather dataset.

Thanks to the Outliergram it is possible to catch the shape outliers or what is the same, the curves that exhibit an anomalous form in the sample. In Figure 4.2 it is displayed the the outliergram for the previous dataset and as explained in the previous sections a parabola is calculated to show that the closest points to it are the ones having a more normal shape, while the ones that are further present a not so similar form. The dashed parabola is the outlier detector, so all the points that lie below it are considered shape outliers. The two observations considered as outliers, have been represented in Figure 4.2(b) to prove the experiment with the outliergram is working. As we can see, this curves are flatter, having not so cold winters and small temperature changes along the year.



(a) Outliergram of the temperatures dataset.



(b) Curves detected as shape outliers by the outliergram.

Figure 4.2: Figure 4.2(a) shows the detected shape outliers by the outliergram, which can be seen represented as curves in Figure 4.2(b).

The points that are over the upper parabola are not necessarily considered outliers due to the calculation of the parabola is an approximation of the Modified Band Depth and Modified Epigraph Index each point should have to show the most typical shape, independently of the magnitude outliers that mainly the MBD can detect.

4.2 DD Plot

For the experiments related with the DDPlot the samples taken in the Artic station are not used, as I just want three different types of data. The Artic sample was rejected as this one was the less numerous and it gave us the least information. The experiments explain all the possible outputs and consequences this plot can have. The main data used in the tests corresponds to all the observations of the Atlantic weather. In the different experiments, I swapped the different distributions used in each axis to compare with the Atlantic sample.

The following graph (Figure 4.3) represents the samples used in the experiments thanks to labels indicating the three different climates:

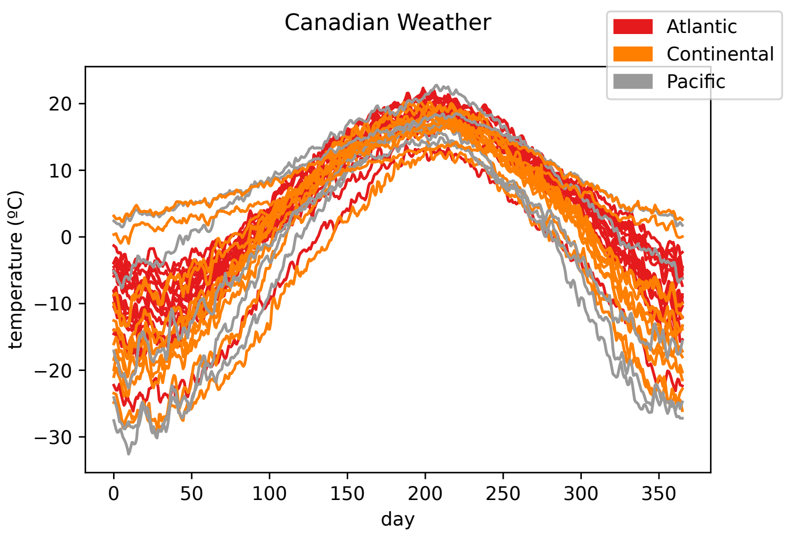
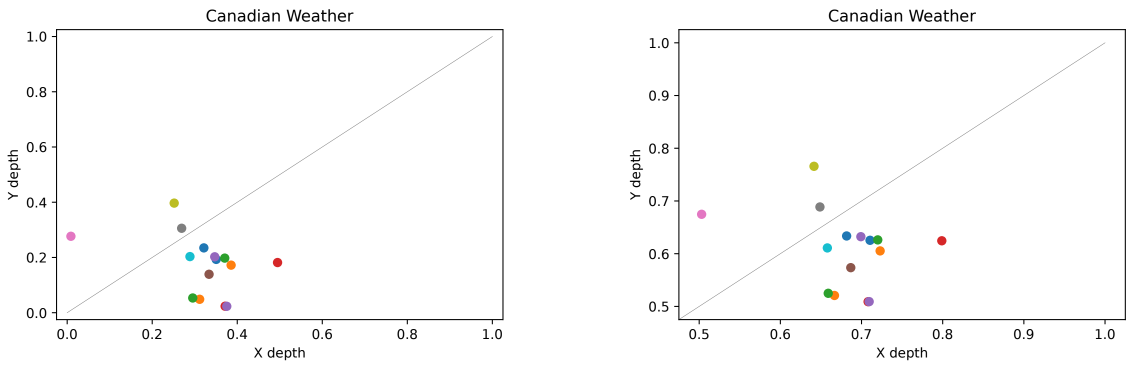


Figure 4.3: Atlantic, Continental and Pacific temperature functions of the Canadian Weather dataset.

The next two graphics represent the comparison of the depth of each Atlantic sample in the two distributions formed respectively by the Pacific (X axis) and Continental (Y axes) temperature functions.



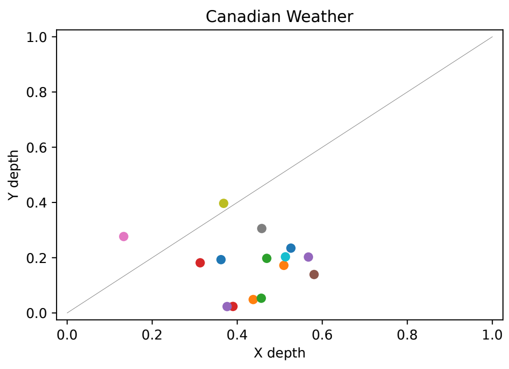
(a) DDPlot computed with MBD.

(b) DDPlot computed with Integrated Depth.

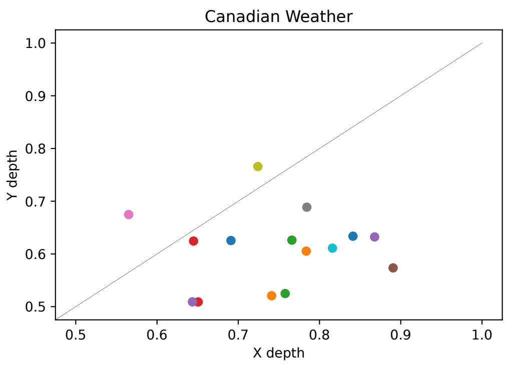
Figure 4.4: These graphs represent the DDPlot of the Atlantic samples compared with the Pacific samples (X axis) and Continental samples (Y axis). The difference between the graphs is the depth method used.

As it can be seen in spite of the fact I am using different depth measures to create the graphics the conclusions are similar, almost all of the instances of our Atlantic data look more similar to the pacific distribution. This is known because the points are located below the grey line, what means the have a higher depth with this distribution than the other one. The more points below and the higher the depths they have the more similar they are to the x axis distribution. This can also be useful to know how similar is our data to the two distributions or perhaps find outliers.

These two DDPlot representations are the comparison of the depth of each Atlantic sample with itself (X axis) and continental (Y axes) temperature functions.



(a) DDPlot computed with MBD.

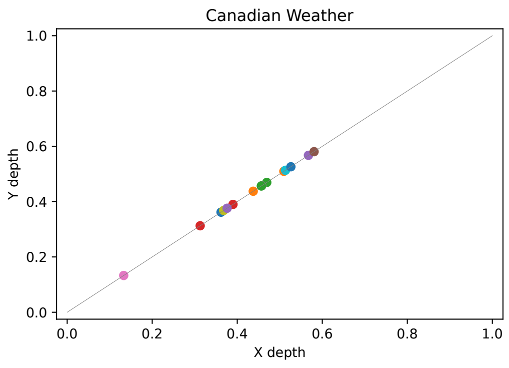


(b) DDPlot computed with Integrated Depth.

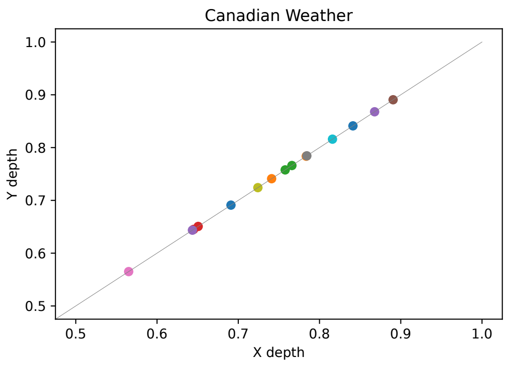
Figure 4.5: These graphs represent the DDPlot of the Atlantic samples compared with the Atlantic samples (X axis) and Continental samples (Y axis). The difference between the graphs is the depth method used.

This case of use is not that common, but it can result useful to find outliers too. The results of the experiment make sense, as almost all the points show high depths with its distribution and only two have a bit higher depth in the continental dataset.

These two DDPlot representations are the comparison of the depth of each Atlantic sample with itself in both axis.



(a) DDPlot computed with MBD.

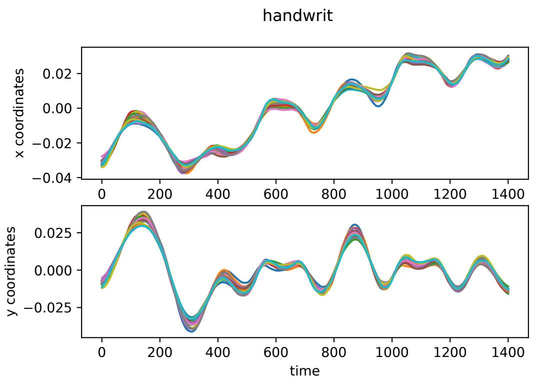


(b) DDPlot computed with Integrated Depth.

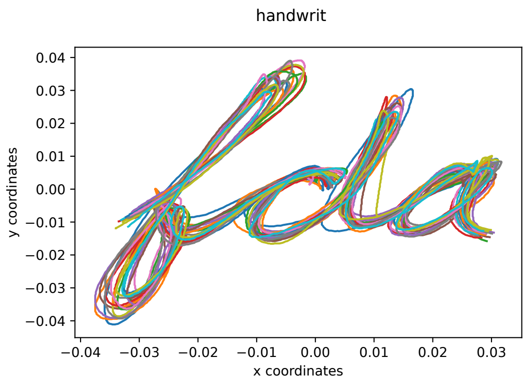
Figure 4.6: These graphs represent the DDPlot of the Atlantic samples compared with with itself in both axis. The difference between the graphs is the depth method used.

4.3 Parametric Plot

One good example to test the proper functionality of our ParametricPlot is the handwrit dataset from the fda package (available in CRAN). It was created by J. Ramsay and represents the x and y coordinates obtained as result after writing 20 times the word fda. In Figure 4.7(a) it can be seen the representation of the coordinates over time.



(a) Representation of coordinates along time.

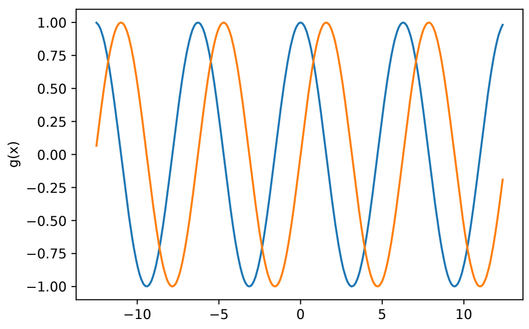


(b) Parameterization of coordinates x and y.

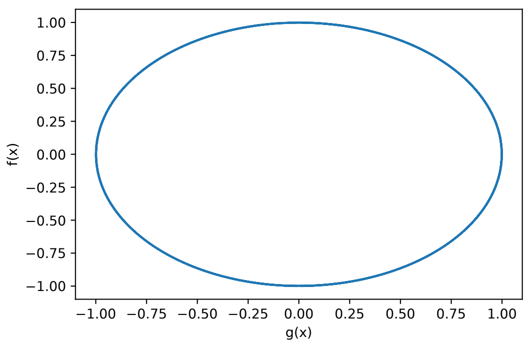
Figure 4.7: Parameterization of word fda.

By parameterizing the coordinates the word fda is obtained, as the x axis represents the x coordinates and the y axis y coordinates. With the graphic represented in Figure 4.7(b), the time is ommited and it gives us a clear example of what the ParametricPlot can do.

The next example is shown in Figure 4.8 and uses the next functions: $f(x) = sin(x)$ and its derivative, $f'(x) = cos(x)$.



(a) Representation of f(x) and f'(x).

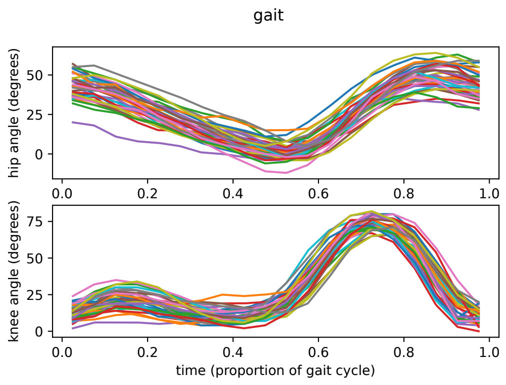


(b) Parametric plot representing (f(x), f'(x)).

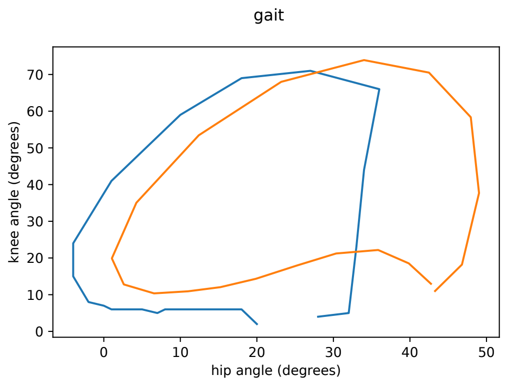
Figure 4.8: Representation and parametric plot of the equations above.

Due to the form of this trigonometric functions and how they interact with each other, when they are plotted with its derivatives they tend to form loops, in this case creating a circle.

One of the other uses the ParametricPlot has, is comparing different parameterized curves. An example shown in Functional Data Analysis [4] is done with a dataset containing the angles made by the hip and knee of different children during its gait cycle. This two samples of curves, that represent hip and knee angles changing during the gait cycle, don't show us two much information when represented alone. In Figure 4.9 there is a comparison of the parameterization of the mean of both samples (orange) and one child (blue).



(a) Representation of hip angle and knee angle curves over the gait cycle.



(b) Parametric plot representing the parameterization of the average hip and knee angle (orange), and the parameterization of any instance (blue).

Figure 4.9: Representation of gait dataset and parametric plot representing the parameterization of the average hip and knee angle, and the parameterization of any instance.

Thanks to this graphic (Figure 4.9(b)), with a quick glimpse, anomalies in the way of walking of a child can be detected in order to give him a special treatment if needed. The child was selected using the Outliergram to see how a anomalous shape can exhibit strange behaviour in its parameterization. In Figure 4.10 there is a parameterization of all the curves of the sample.

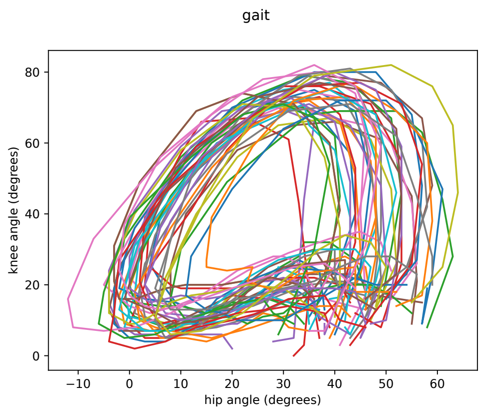
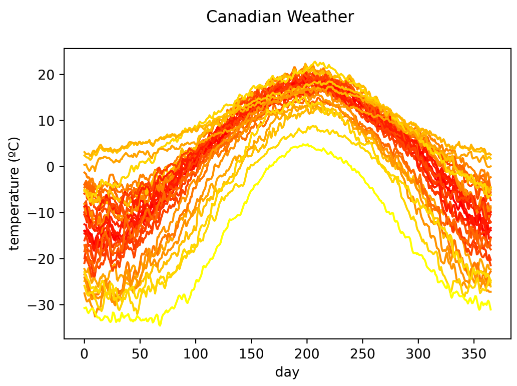


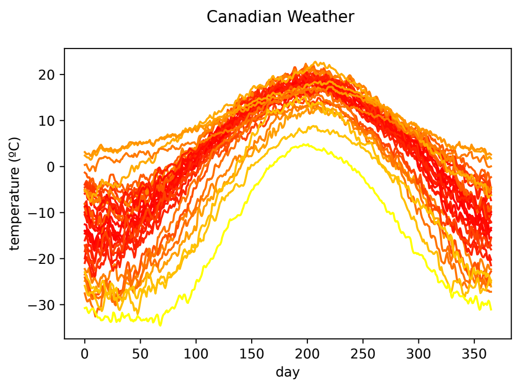
Figure 4.10: Parameterization of the gait dataset.

4.4 Graph Plot with gradient of colors

In the GraphPlot it was added an option to plot the a dataset of curves or surfaces with a gradient of colors depending on a list of parameters. As it may be expected, one of the most important measures that can be done in FDA are depths. In the next images (Figure 4.12), it can be seen the same plot as the one done in Figure 4.1, but using the Integrated Depth and Modified Band Depth to plot each curve with a different color depending on the depth. The higher the depth is, the more intense the color. Nevertheless, this depends on the selected colormap and it can be also inverted.



(a) GraphPlot using Integrated Depth.



(b) GraphPlot using Modified Band Depth.

Figure 4.11: This graphs represent temperatures in the canadian station showing a color gradient depending on functional depths.

Another interesting example would be using the already mentioned Modified Epigraph Index, which doesn't actually measure the centrality of a curve or surface but the average time it stays under other instances. This can be seen in Figure 4.12, as the curves with a more intense color are the ones that have more curves above for more time.

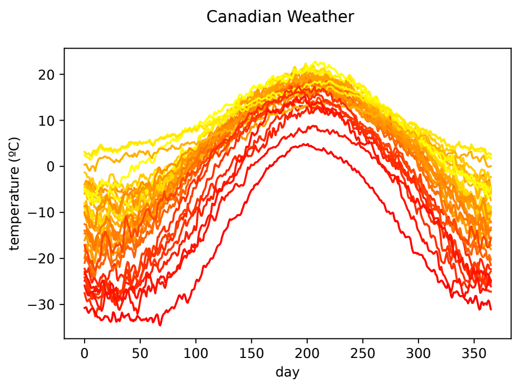


Figure 4.12: Graph representing temperatures in the canadian station showing a color gradient depending on MEI.

4.5 Multiple Display

The Multiple Display functionality allows us to combine graphics and interact with them thanks to widgets, clicking points or hovering them. The next image (Figure 4.13) shows a clear example.

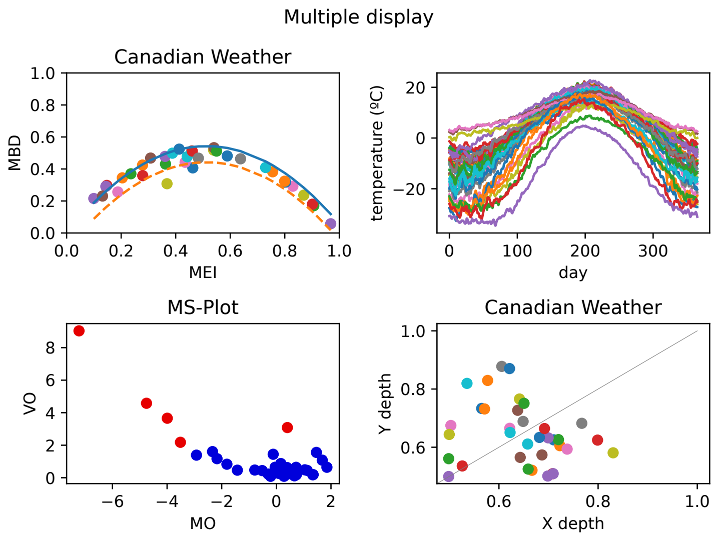


Figure 4.13: Multiple Display containing Outliergram, GraphPlot, Magnitude Shape Plot and DDPlot.

This first case of the multiple display functionality is the most basic appearance possible, as there are no widgets created, none of the points of the graphs have been clicked and neither one of them is being hovered. In Figure 4.14 as you may observe, one of the instances has been selected by clicking. As it has been clicked, the rest of the instances had incremented their transparency in every graph.

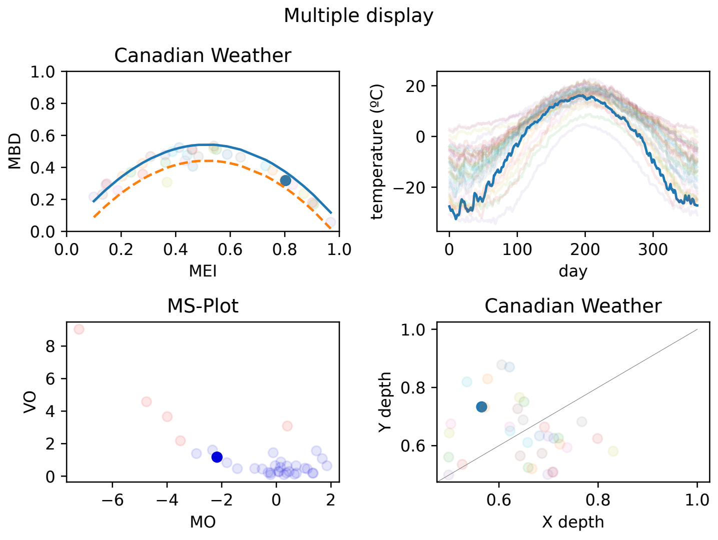


Figure 4.14: Same graph as Figure 4.13 but one point has been selected.

From the situation above I will explain what could happen as it can only be seen by interacting directly with the tool or seeing a video. If the same point was clicked, the original graph of Figure 4.13 would be restored but if any other point was clicked this one would return to its original form and the first point clicked would reduce its opacity in all the plots. This is a good form of interacting with the graph as there is an easy way to return to the original state.

The following multiple display case only present two graphs, an Outliergram and a GraphPlot, together with two sliders that have as criteria Modified Band Depth and Integrated Depth. The slider presents 35 different options (one for each curve represented). In Figure 4.15 there is an example of the widget being clicked.

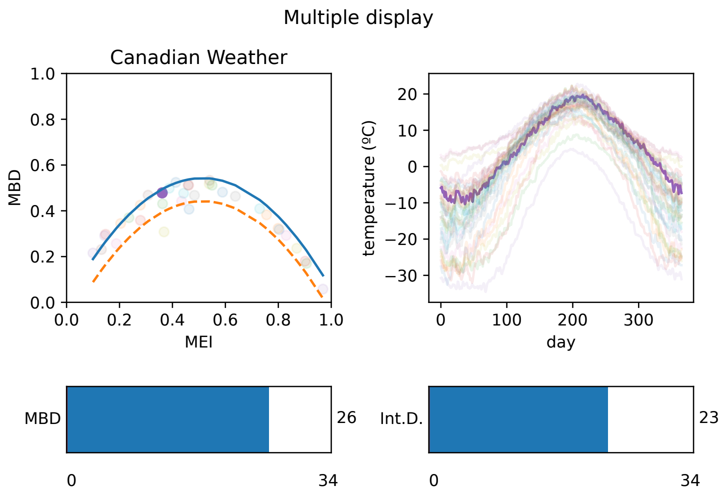


Figure 4.15: Multiple Display containing Outliergram, GraphPlot and two sliders representing MBD and Integrated Depth.

As it appears in the picture above, the user has used the widgets to select an instance. We can see the values obtained make sense, as in the outliergram the MBD is represented in the Y axis and as it appears in the first widget, the point selected and the widget have a relatively high value. It is important to highlight that even if the selected widget was the one that used Modified Band Depth, the other one was also updated to the corresponding value of that point in its widget criteria. From this point we could continue using the widgets to select other curves and they would continue having congruent values. It would also be valid to click a point instead of using the widget again as they activate the same functionality.

Figure 4.16 contains the last experiment done with the interactive module. The graphics show an example of the outcome of hovering a graph, which shows us some extra information of a point. In the outliergram, our user has tried to get extra information about the curve that presents the highest Modified Band Depth. To do this the user has hovered the graphic and obtained the corresponding MBD and MEI values (0,54 and 0.53, respectively). Besides he also can see that this point represents the curve 15 of our dataset. This information is very useful if users want to get information fast about any instance. Besides, other forms of getting a specific curve are way more difficult for a common user, with this the users job is speeded up so they get the best results as soon as possible.

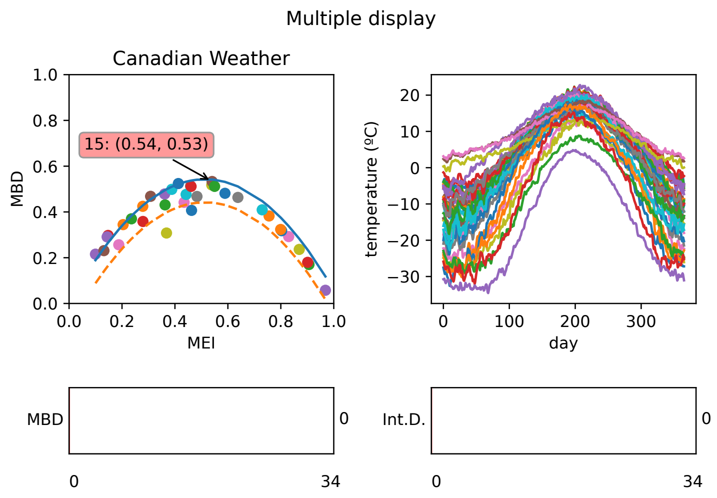


Figure 4.16: Hovering example with a Multiple Display.

5 Conclusions and future work

Working in this project has resulted to be a very fulfilling experience due to various reasons. The first is that when I searched for a final degree work I was very interested on the idea of working in a team with different developers and people from multiple knowledge areas. This really helped me to learn how it would be working in an open source software project and gave me experience using advanced tools like GitHub to upload code, automatically test it or create pull requests and branches. Furthermore, I feel like I contributed to the library in a very important part of it, which is the visualization of the data, giving users more tools to use and have a better experience with the package.

I am also satisfied with the changes developed as I think the library is easier to use for the newcomers thanks to the homogenization and standarization of the code corresponding to the visualization section. I also feel that my coding style and documentation skills have improved drastically, as now I am used to following strict coding rules, which will help me in my future profesional or research projects, as it will be easier for the rest of people and even for me to understand my code. It also helped me to discover an unknown world previously for me, Functional Data Analysis. I really got interested to it due to the extense applications it can have in different fields like biology, engineering, economics, ect.

I think that the visualization area is very extense and in the future it could receive new features or extra tools. One of the objectives of the project, was to make sure the software was easily scalable and future developers could use the structure I designed to continue adding new plots. An interesting new graphic that could be added is the LasagnaPlot [40], which is a different representation used to view how the different curves of a sample are aligned. It would be interesting to see how this plot (heatmap) could interact with the interactivity module and trying to search for new functionality to be added. This functionality could help us interact with other kinds of graphics and try to make our interactivity module compatible with as much different types of plots as possible. Apart from this, a logical expansion of the ParametricPlot would be extending this functionality to three dimensions so it would be possible to parameterize curves in a 3 dimensional space (functions with type $f : \mathbb{R} \longrightarrow \mathbb{R}^3$). In the design aspect, late in the project it was also considered implementing the Composite design pattern [41] for the visualization module, in which the MultipleDisplay object would be also a BasePlot. It was not finally implemented in this project because there weren't many use cases where it would be used and the cost of developing it was high, but in the future it could be added. Another interesting new feature, would be generalizing the text that can be added to the plots and subplots (titles, labels, etc.) so that it can be controlled by the user. To end with future work, I should highlight that in a few years new widget libraries may appear or Matplotlib could be expanded, so it would be great to look for the possibility of having more types of widgets available that work in every Python backend, as the current ones.

To sum up, I feel like during this project I have managed to apply multiple competences that I acquired while studying my degree like coding, statistics, software engineering or machine learning. I also think that thanks to it I could expand my knowledge beyond computer science and explore other interesting areas like FDA.

Bibliography

[1] C. R. Carreño, "scikit-fda: A python package for functional data analysis.," *III International Workshop on Advances in Functional Data Analysis*, 2019.

[2] P. Virtanen, R. Gommers, and S. et al., "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[3] "Github: Gaa-uam / scikit-fda." `https://github.com/GAA-UAM/scikit-fda`, 2018.

[4] J. Ramsay and B. Silverman, *Functional Data Analysis*. Springer-Verlag New York, 2005.

[5] F. Ferraty and P. Vieu, *Nonparametric Functional Data Analysis*. Springer-Verlag New York, 2006.

[6] F. S. e. a. J. Goldsmith, "refund: Regression with functional data." `https://cran.r-project.org/web/packages/fda.usc/index.html`, 2012.

[7] S. G. J.O. Ramsay and G. Hooker, "fda: Functional data analysis." `https://www.psych.mcgill.ca/misc/fda/software.html`, 2017.

[8] J. D. Tucker, "fdasrvf: Elastic functional data analysis." `https://cran.r-project.org/web/packages/fdasrvf/index.html`, 2014.

[9] K. Hornik, "The comprehensive r archive network." `https://cran.r-project.org/`.

[10] C. R. Carreño, A. Suárez, J. L. Torrecilla, M. C. Berrocal, P. M. Manchón, P. P. Manso, A. H. Bernabé, D. G. Fernández, Y. Hong, and P. M. R.-P. Eyriès, "Gaa-uam/scikit-fda: Version 0.5," Dec. 2020.

[11] A. Hernando, "Development of a python package for functional data analysis. depth measures, applications and clustering," 2019.

[12] "scikit-fda documentation." `https://fda.readthedocs.io/en/latest/index.html`, 2019.

[13] P. T. Inc., "Collaborative data science," 2015.

[14] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.

[15] L. Wilkinson, *The Grammar of Graphics*. Springer-VerlagBerlin, Heidelberg, 2005.

[16] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[17] C. Harris and S. v. d. W. e. a. K. Jarrod and, "Array programming with NumPy," *Nature*, vol. 585, 2020.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[20] M. Waskom, O. Botvinnik, D. O'Kane, P. Hobson, S. Lukauskas, D. C. Gemperline, T. Augspurger, Y. Halchenko, J. B. Cole, J. Warmenhoven, J. de Ruiter, C. Pye, S. Hoyer, J. Vanderplas, S. Villalba, G. Kunter, E. Quintero, P. Bachant, M. Martin, K. Meyer, A. Miles, Y. Ram, T. Yarkoni, M. L. Williams, C. Evans, C. Fitzgerald, Brian, C. Fonnesbeck, A. Lee, and A. Qalieh, "mwaskom/seaborn: v0.8.1 (september 2017)," Sept. 2017.

[21] M. O. de la Fuente, "Functional data analysis using fda.usc package." https://rpubs.com/moviedo/fda_usc_introduction.

[22] H. L. Shang, "rainbow: An r package for visualizing functional time series." https://journal.r-project.org/archive/2011/RJ-2011-019/RJ-2011-019.pdf, 2011.

[23] H. L. Shang, "rainbow: Bagplots, boxplots and rainbow plots for functional data." https://cran.r-project.org/web/packages/rainbow/index.html, 2011.

[24] "tidyfun." https://github.com/tidyfun/tidyfun.

[25] . M. G. Fraiman, R., "Trimmed means for functional data," *Nature Methods*, vol. 10(2), pp. 419–440, 2001.

[26] R. J. López-Pintado S., "On the concept of depth for functional data," *Journal of the American Statistical Association*, vol. 104, pp. 718–734, 2009. Link.

[27] J. P. R.Y. Liu and K. Singh, "Multivariate analysis by data depth: Descriptive statistics, graphics and inference (with discussion)," *Ann. Statist*, vol. 27, pp. 822–831, 1999.

[28] R. Liu and K. Singh, "A quality index based on data depth and multivariate rank test," *Journal of the American Statistical Association*, vol. 88, 1993.

[29] A. W. D. Kosiorowski, M. Bocian and Z. Zawadzki, "ddplot: Depth versus depth plot." https://rdrr.io/cran/DepthProc/man/ddPlot.html.

[30] M. O. de la Fuente and M. F. Bande, "classif.dd: Dd-classifier based on dd-plot." https://rdrr.io/cran/DepthProc/man/ddPlot.html.

[31] A. Arribas-Gil and J. Romo, "Shape outlier detection and visualization for functional data: the outliergram," *Biostatistics*, vol. 15, pp. 603–619, oct 2014. Download.

[32] S. Y. and G. M. G.., "Functional boxplots," *Journal of Computational and Graphical Statistics*, vol. 20, pp. 316–334, 2011. Link.

[33] R. J. López-Pintado S., "On the concept of depth for functional data," *Journal of the American Statistical Association*, vol. 104, pp. 1679–1695, 2009. Link.

[34] R. J. López-Pintado S., "A half-region depth for functional data," *Computational Statistics Data Analysis*, vol. 55, pp. 1679–1695, 2011. Link.

[35] B. W. G. van Rossum and N. Coghlan, "Pep 8 – style guide for python code," jul 2001.

[36] D. Goodger and G. van Rossum, "Pep 257 – docstring conventions," may 2001.

[37] "ipywidgets." https://ipywidgets.readthedocs.io/en/latest/, 2021.

[38] J. Nielsen, *Usability Engineering: Amazon.de: Jakob Nielsen: Englische Bücher*. 2012.

[39] S. A. S. LaZerte and N. Brown, "Package 'weathercan'." https://cran.r-project.org/web/packages/weathercan/weathercan.pdf, jan 2021.

[40] J. Goldsmith, “Visualization.” `https://tidyfun.github.io/tidyfun/articles/x04_Visualization.html`, may 2020.

[41] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 ed., 1994.

[42] J. L. G. van Rossum and Langa, “Pep 484 – type hints,” sep 2014.

[43] I. L. e. a. R. Gonzalez, P. House, “Pep 526 – syntax for variable annotations,” aug 2016.

APPENDICES

A Tools used

A.1 Wemake-python-styleguide

To succeed in our mission of creating a standard, understandable and efficient code, during this year we started following wemake-python-styleguide (WPS). This styleguide is a flake8 plugin which has as its goal to enforce the programmers of the library to follow the best code practices. The plugins and libraries used in the linter (tool used to analyze code and detect any kind of error) are specified in the setup.cfg file. This file also contains exceptions and other additional information related with linting. For the installation of the WPS tool, a new post in the wiki was created (this was done by other student in the project, Pedro Martín Rodríguez) and it will be useful for future developers of the tool, as GitHub does automatic style WPS tests every time a pull request is updated. The main goals of using this linter are finding style errors, bugs, revising complexity of methods or classes to make a more maintanable and scalable code and making developers use good coding practices.

This are the following tools used in WPS:

- **Flake8** is a common linting tool used in python to avoid errors, correct style errors and write a higher quality code, which is in our case very important. It makes sure the standards already mentioned related with coding and documenting are properly followed.
- **isort** is a library specialized on making sure the the imports are divided in groups, sorted alphabetically and have the proper spaces between them. This is done for a better comprehension of the code and its imports.
- **Mypy** is a static type checker, which uses the Python3 annotations of PEP 484 [42] and PEP 526 [43]. Even if Python has the "advantage" of not having to indicate the type of the variables, in big and complex codebases like ours, it really helps. For such a big project like scikit-fda it gives multiple advantages like improving the comprehension of the library for other developers or the users of the library, better debugging, reducing the effort made by future developers and catching errors faster. As the arguments and outputs of the different functions are typed, if a function receives an incorrect type, the error will be detected before the code is executed.

There are two main ways to type our code, nominal and structural subtyping. The first is the most common and the most used during the project, as it checks the compatibility of types based on the hierarchy of classes. The second one is based on classes sharing the attributes and methods specified. To use structural subtyping it is common to use predefined Protocols (which are classes that define the methods and attributes that need to be shared) or define our own protocols, giving us a lot of flexibility in typing.

A.2 Testing

Due to the high amount of potential collaborators an open-source software project has it is basic to have a good testing framework. In our case there are two testing frameworks, the most famous in Python: *unittest* and *pytest*.

- ***unittest***: this package is inspired on JUnit, a testing framework for Java programming language. Its way of testing is (as you may imagine by its name) unitary testing. Unitary tests, are used to check the correct functionality of elementary parts of our code (units). As the software is object-oriented, these basic elements could be the classes designed. To test all of our code *unittest* provides us with different types of tests: test cases, which are the most fundamental ones; test suites, that are collections of test cases or test suites, and used to test different functionalities in one execution; test fixtures, which are used to do elementary actions needed to do the tests (creating and populating a database or starting a server) and test runners which are used to combine all of them and get a final result for the user.
- ***pytest***: despite almost all of the test had been developed with the other framework and it has its own way of executing all the test, in this project I have been using *pytest* to execute all the tests as it has better reporting features. These features include complete information about errors that have happened when testing (giving us information to trace them), the ability of being combined with *unittest* and an extense collection of plugins.

To make sure that all the project has been properly tested, I have been using **Codecov**, a tool that realizes code coverage tests, giving a visual measurement of what parts of our code are being covered by our test suites. Thanks to this, it is known which parts need more tests and the total code coverage. These calculations are automated in our GitHub repository when making pull-requests.

A.3 Documentation

In order to obtain the best documentation of the project, I used the tool Sphinx, a Python document generator under the BSD license. Thanks to this tool our code and documentation is transformed from reStructuredText into other formats like PDF or HTML. This is very useful as it allows us to display in the projects API examples explaining how the classes work. Moreover, as the library can result difficult at first for beginners, they can check the code from the website thanks to Sphinx and even download it as a Python file or a notebook. To review this online examples, I use doctests that revise the docstrings in order to view if the examples they have are updated with the current code, so the documentation is coherent with the changes.

A.4 GitHub

The tool used in the project to store and develop our code is GitHub. This code repository allows users to work collaboratively using the Git version control system. It allows the creation of private or public repositories depending on the aim of the project. In the second case, any person can create pull request to try to add new features to your code, what makes this a perfect place to develop an open-source software. Besides, GitHub can use external bots that are activated when new code is uploaded to a pull request. In our case, the repository has multiple bots to test the test coverage of the code (Codecov), do automated style tests (wemake-python-styleguide) and ensure continuous integration (Travis CI).

B Usability test

In this section of the appendix, I will explain the usability test done to one of the developers of the library, Carlos Ramos. The technique used in this test is Thinking Aloud, in which the user does specific chores defined by the tester. While the user is doing that, he is supposed to express how he is feeling with the software, possible doubts or questions about the program. It is important to take into account when doing this test, who is the subject and what can be expected of them, to extract the best conclusions of the process.

The user will be analyzing the visualization module and was asked to do the next chores:

- Download the branch containing the upgraded module.
- Create a notebook and represent the Canadian weather dataset using the new functionality (GraphPlot).
- Create the outliergram that represents the temperature functions of the Canadian weather dataset.
- Create an interactive display combining the outliergram and a representation of the dataset.
- Interact with the outliergram selecting the point with the lowest Modified Band Depth, that can be identified thanks to the axes.
- Hover the points that are considered shape outliers to get their position and id.
- Pick the outlier points with the mouse in order to check that the shape they expose is atypical.

During the test, I took down all what he said and reached the following conclusions:

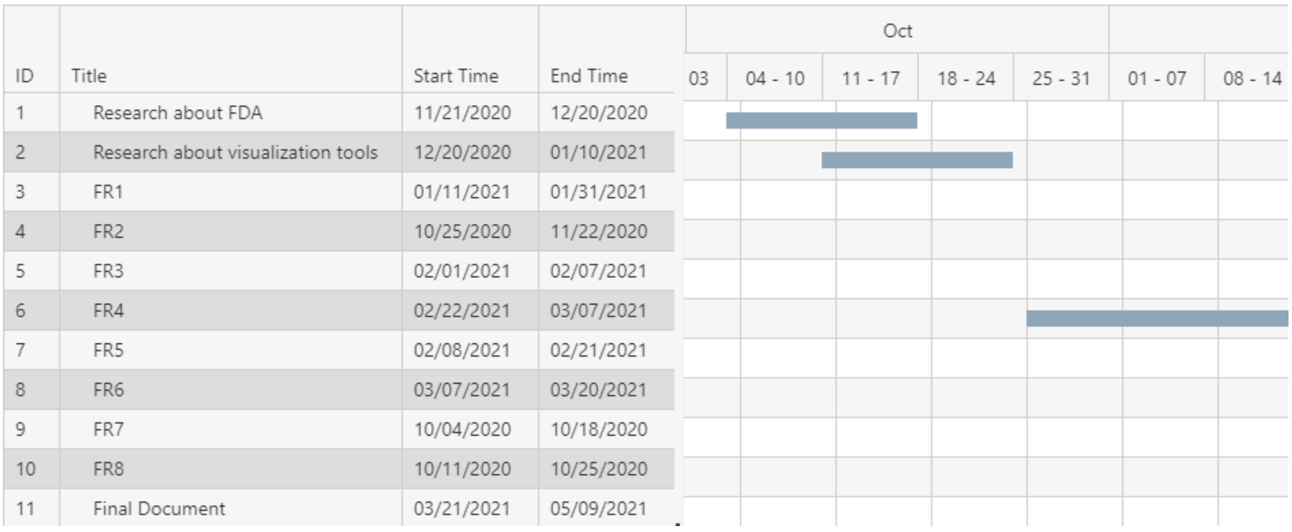
- The user experiences certain discomfort with Matplotlib and the fact that when creating the figure when initializing a BasePlot they are always plotted. Anyway this can't be solved by us as this is a problem of the library.
- He is satisfied with the colors and how the Outliergram object is displayed. He also is satisfied with the precision of the parabolas computed.
- He feels comfortable with the homogeneous way of creating and plotting visualization objects.

- He suggests the possibility of adding parameters that can specify the colors of any plot. This has been added to the future work of the project.
- He feels comfortable with the way of interacting with points clicking them and shows no problems using the tool despite being the first time.
- He achieves all the tasks without any problem and feels satisfied with how he can get insights thanks to the Multiple Display.
- He finds useful for users the possibility to get in a fast way the id of a sample just by hovering it.
- He made suggestions to the titles and labels of the different subplots. Thanks to this I modified it and decided to have as title the name of the dataset.

C

Gantt Chart

Gantt Chart created for the organization of the project. In it I included the initial research studies done about Functional Data Analysis, to get a good basis about this area of statistics, and about the most advanced visualization and interactive tools that exist and could fit in the scikit-fda project. Afterwards, I added all the functional requirements that needed to be implemented. Finally the time expected for the creation of the final version of the document, considering that during development of the Functional Requirements previous temporary documents where created.



ID	Title	Start Time	End Time
1	Research about FDA	11/21/2020	12/20/2020
2	Research about visualization tools	12/20/2020	01/10/2021
3	FR1	01/11/2021	01/31/2021
4	FR2	10/25/2020	11/22/2020
5	FR3	02/01/2021	02/07/2021
6	FR4	02/22/2021	03/07/2021
7	FR5	02/08/2021	02/21/2021
8	FR6	03/07/2021	03/20/2021
9	FR7	10/04/2020	10/18/2020
10	FR8	10/11/2020	10/25/2020
11	Final Document	03/21/2021	05/09/2021

Figure C.1: Gantt Chart.

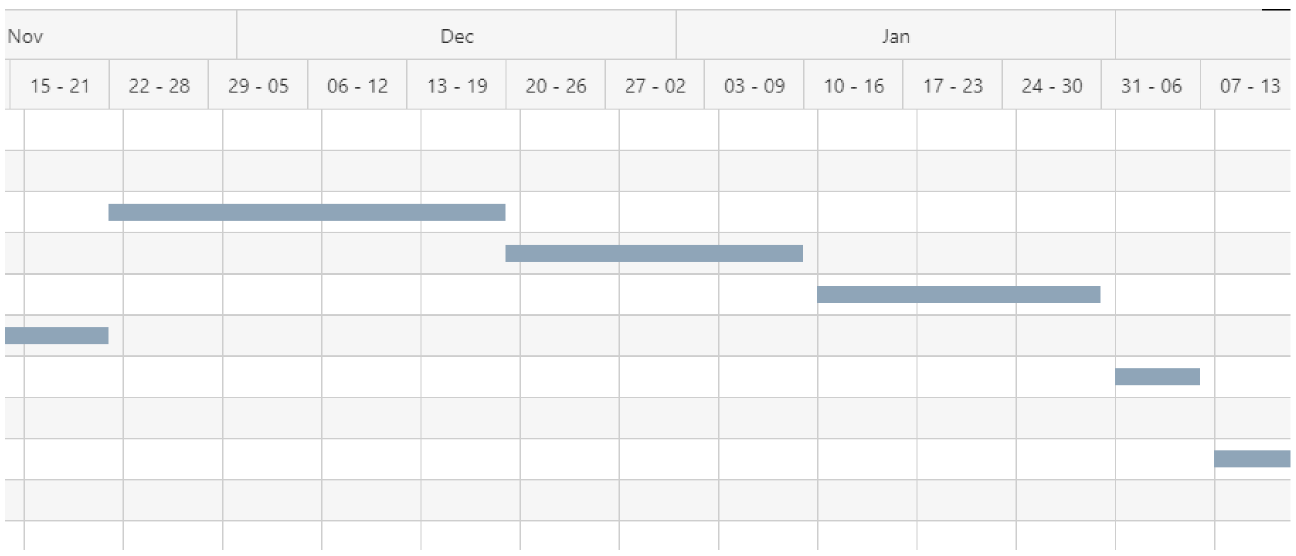


Figure C.2: Gantt Chart.

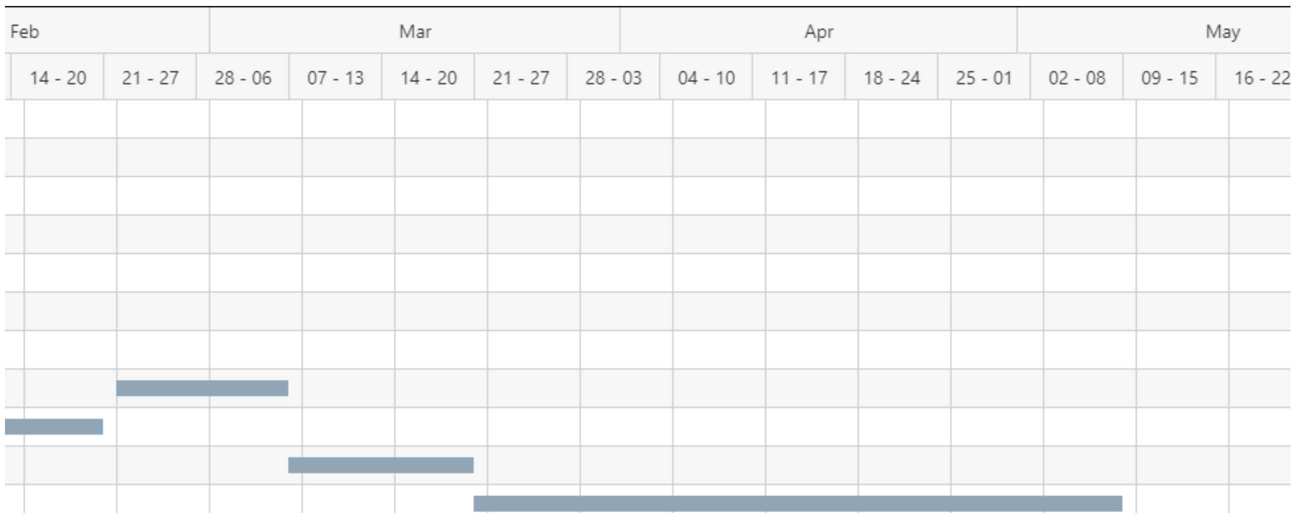


Figure C.3: Gantt Chart.

D

Notebooks

This section contains the notebooks developed for each of the different functionalities created in the project. They contain annotations and simulations done with the new tools of the package.

OutliergramExamples

May 15, 2021

[1]:
```
from skfda import datasets
from skfda.exploratory.visualization import Outliergram

import matplotlib.pyplot as plt
```

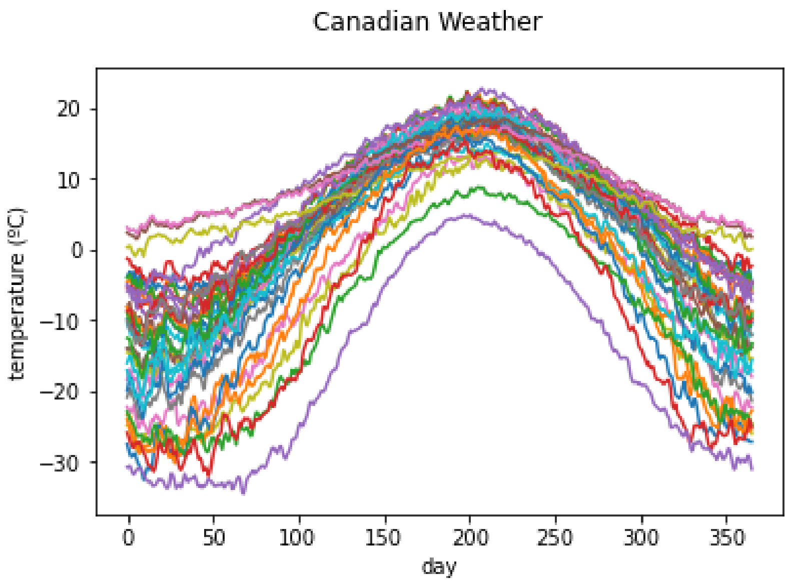
First, the dataset is loaded. In this case, the Canadian Weather dataset from package 'fda' in CRAN is selected. In the experiment, the outliergram cointains the curves representing the temperatures of different weather stations along the year. There are 365 measures, one for each day of the year.

[2]:
```
dataset = datasets.fetch_weather()
fd = dataset["data"]
fd_temperatures = fd.coordinates[0]
```

In the next graphic it can be observed the functions representing the dataset, with the outliergram our objective is to detect curves with an atypical shape.

[3]:
```
fig = fd_temperatures.plot()
```

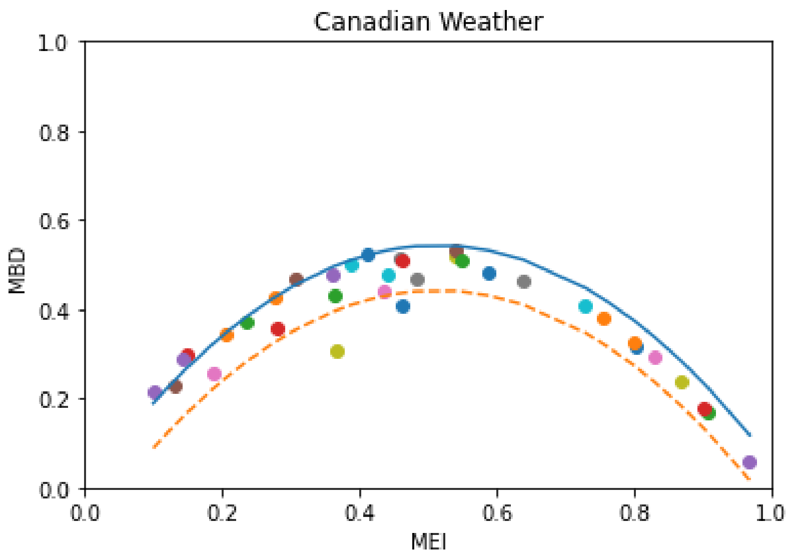
1



There are two shape outliers, the one lying below the dashed parabola.

```
[4]: fig = Outliergram(fd_temperatures).plot()
```

2

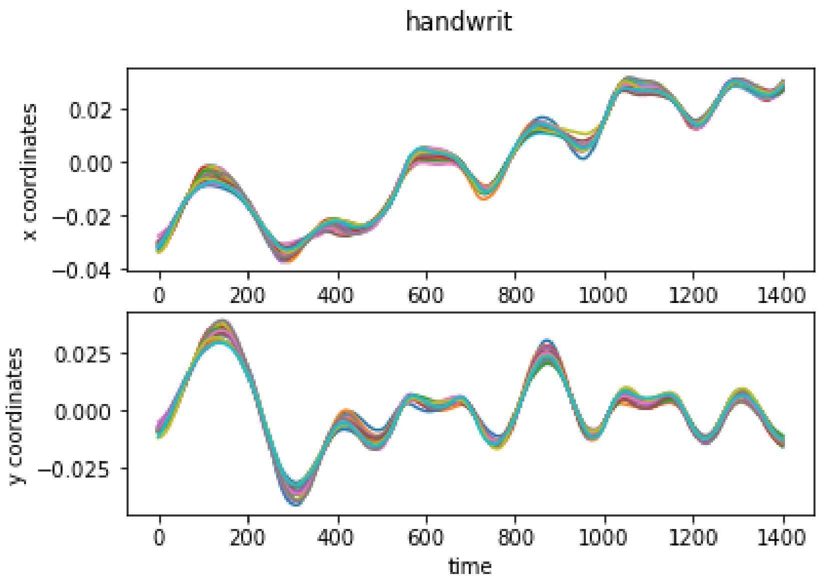


In the next example, the Handwrit dataset also from 'fda' is loaded. It contains the evolution of X and Y coordinates along time while writing the word fda. This dataset was preprocessed by its author so its curves do not represent magnitude outliers. This helps in the detection of shape outliers.

[5]:
```
dataset = datasets.fetch_handwriting()
fd_x = dataset['data'].coordinates[0]
fd_y = dataset['data'].coordinates[1]
```

[6]:
```
fig, axes = plt.subplots(2)
fd_x.plot(ax = axes[0])
fig = fd_y.plot(ax = axes[1])
```

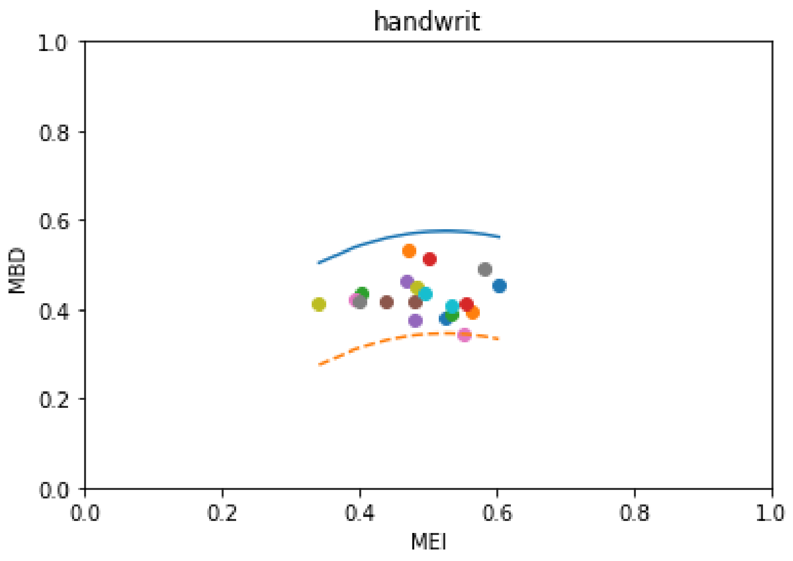
3



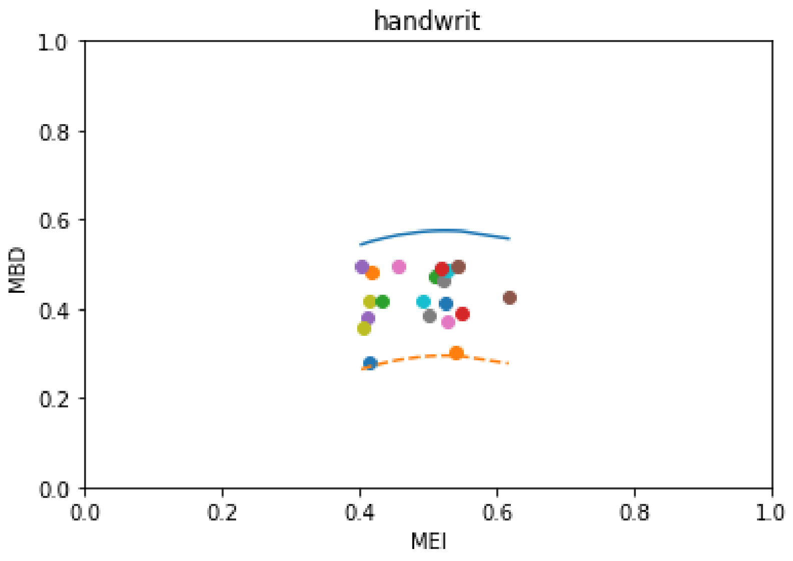
The outliergram in the next cases is much flatter and doesn't have magnitude outliers as the MBD is very high. There aren't any shape outliers in any of the two next outliergrams as no point is under the dashed parabola.

```
[7]: fig = Outliergram(fd_x).plot()
```

4



```
[8]: fig = Outliergram(fd_y).plot()
```



5

6

DDPlotExamples

May 17, 2021

[1]:
```
%matplotlib inline
```

[2]:
```
# Author: Álvaro Sánchez Romero

# sphinx_gallery_thumbnail_number = 2

from skfda import datasets
from skfda.exploratory.depth import IntegratedDepth
from skfda.exploratory.depth import ModifiedBandDepth
from skfda.exploratory.visualization import DDPlot

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

First, the dataset is loaded. In this case, the Canadian Weather dataset from package 'fda' in CRAN is selected. In the experiment, the outliergram cointains the curves representing the temperatures of different weather stations along the year. There are 365 measures, one for each day of the year.

[3]:
```
dataset = datasets.fetch_weather()
fd = dataset["data"]
fd_temperatures = fd.coordinates[0]
```

The dataset is divided in different categories depending on the climate the station belongs to. In this example the sample is divided depending on its climate.

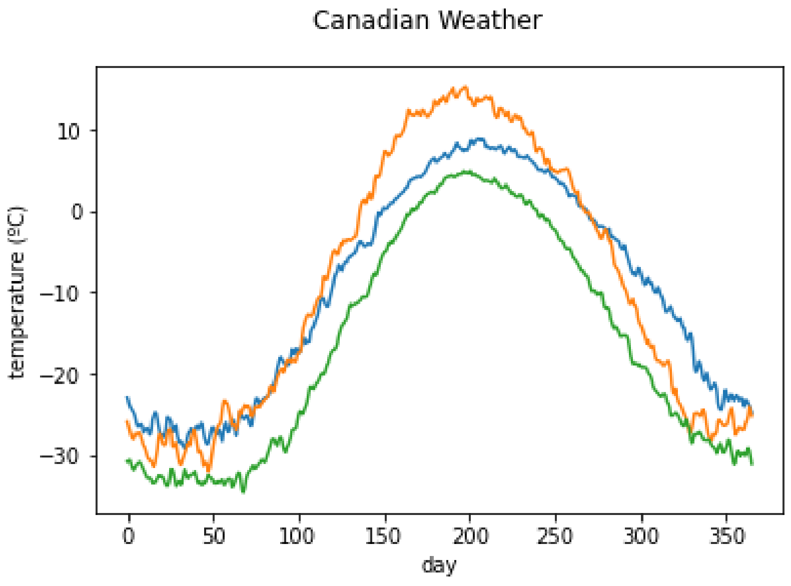
[4]:
```
artic_elements = []
atlantic_elements = []
continental_elements = []
pacific_elements = []
for i in range(fd_temperatures.n_samples):
    if dataset["target"][i] == 0:
        artic_elements.append(i)
    if dataset["target"][i] == 1:
        atlantic_elements.append(i)
    if dataset["target"][i] == 2:
        continental_elements.append(i)
    if dataset["target"][i] == 3:
        pacific_elements.append(i)
```

1

```
fd_temp_artic = fd_temperatures[artic_elements]
fd_temp_atlantic = fd_temperatures[atlantic_elements]
fd_temp_continental = fd_temperatures[continental_elements]
fd_temp_pacific = fd_temperatures[pacific_elements]
```

Representation of the artic temperatures.

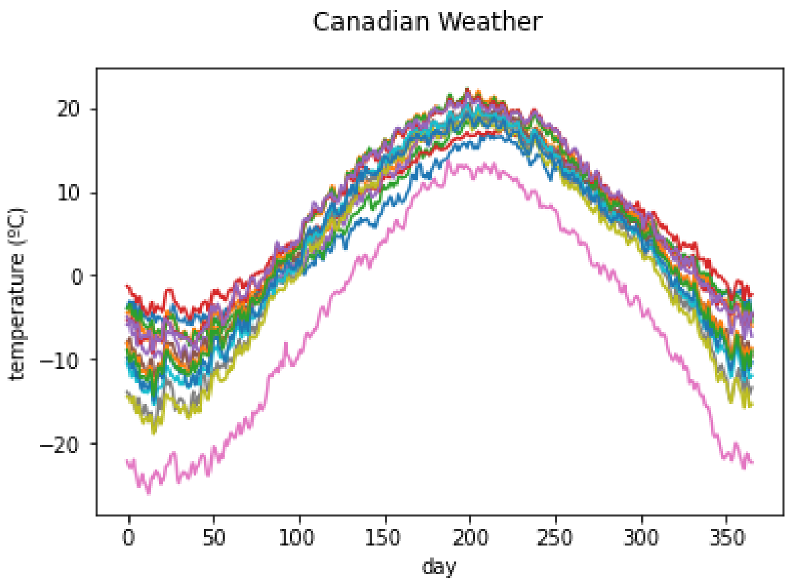
[5]:
```
fig = fd_temp_artic.plot()
```



Representation of the atlantic temperatures.

[6]:
```
fig = fd_temp_atlantic.plot()
```

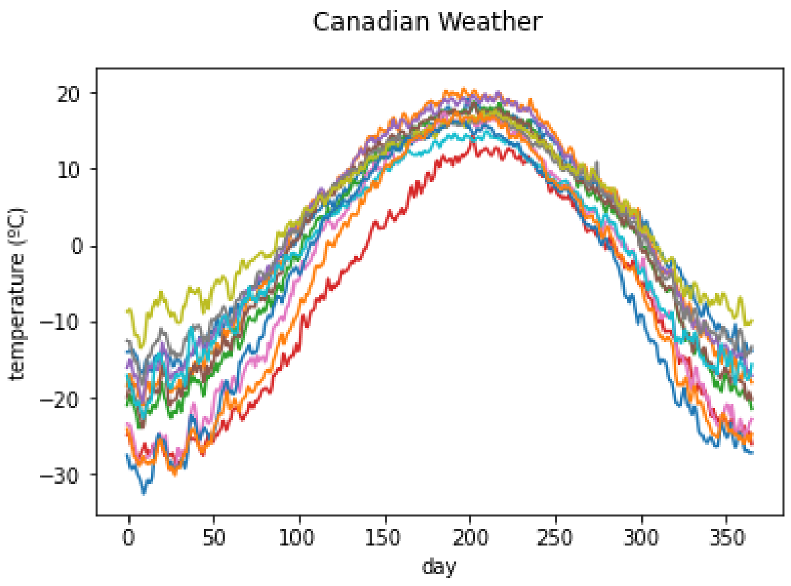
2



Representation of the continental temperatures.

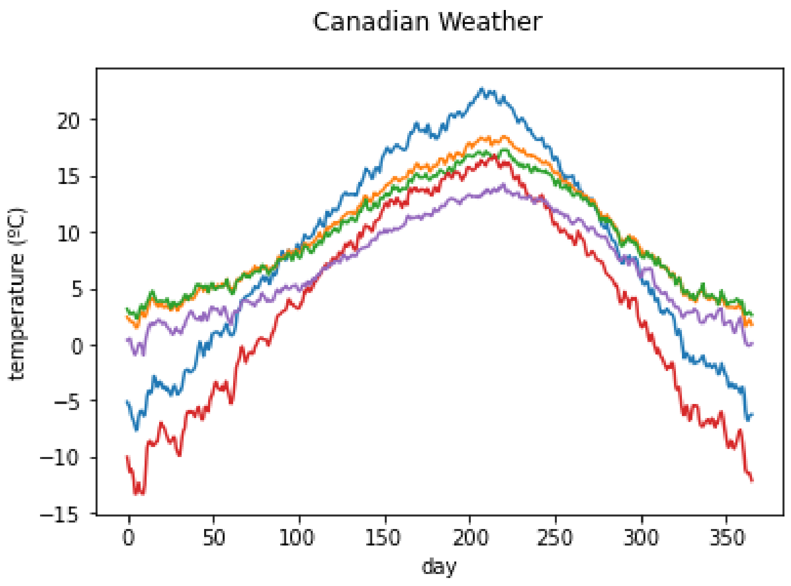
```
[7]: fig = fd_temp_continental.plot()
```

3



Representation of the pacific temperatures.

[8]:
```
fig = fd_temp_pacific.plot()
```



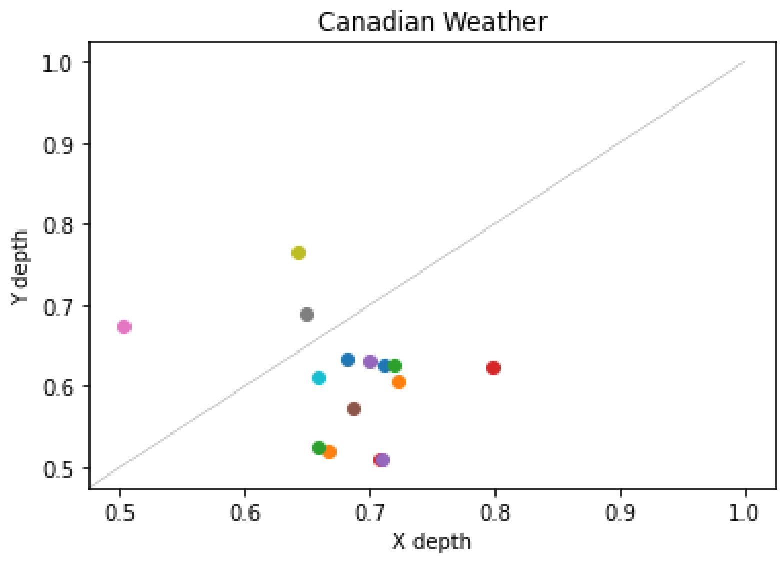
The DDPlot functionality can be tested with different depth methods, the following examples use Integrated Depth and Modified Band Depth, respectively. These two graphics compare the atlantic sample with two distributions, the pacific and continental datasets. Thanks to the DDPlot similarities between the datasets can be detected.

[9]:
```
int_depth = IntegratedDepth()

dd_plot_int1 = DDPlot(fd_temp_atlantic, dist1 = fd_temp_pacific, dist2 =␣
 ↪fd_temp_continental, depth_method = int_depth)

fig = dd_plot_int1.plot()
```

5

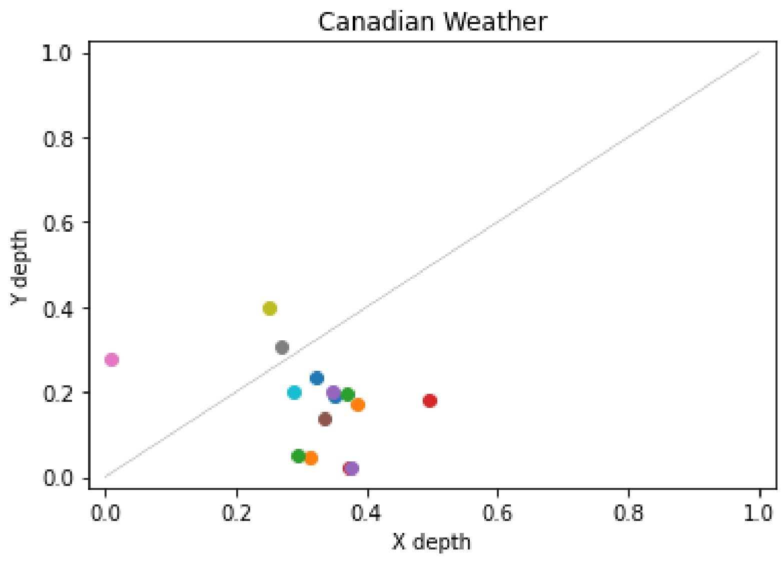


```
[10]: mbd = ModifiedBandDepth()

      dd_plot_mbd1 = DDPlot(fd_temp_atlantic, dist1 = fd_temp_pacific, dist2 =␣
       ↪fd_temp_continental, depth_method = mbd)

      fig = dd_plot_mbd1.plot()
```

6

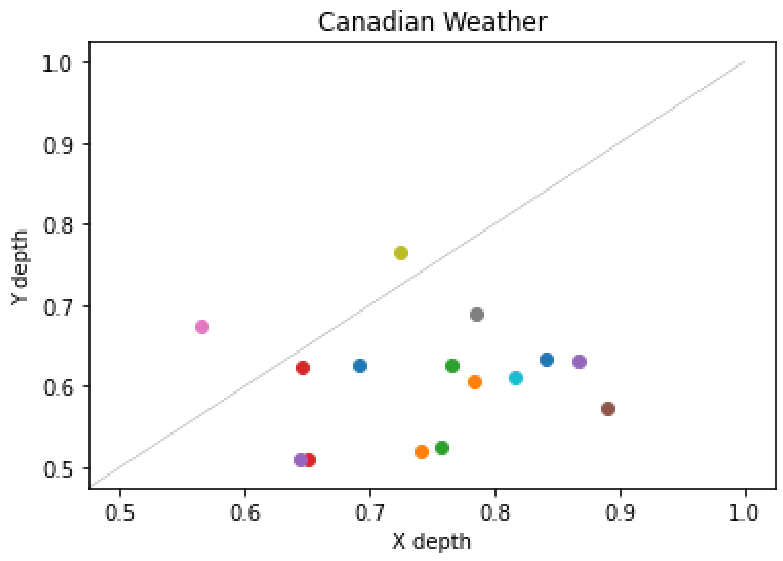


These two graphics compare the atlantic sample with two distributions, itself and continental datasets. As it is logical, almost all of the points are more similar to the atlantic distribution, but two of them have higher depths with the continental one. This might be because they are outliers.

[11]:
```
dd_plot_int2 = DDPlot(fd_temp_atlantic, dist1 = fd_temp_atlantic, dist2 =␣
 ↪fd_temp_continental, depth_method = int_depth)

fig = dd_plot_int2.plot()
```

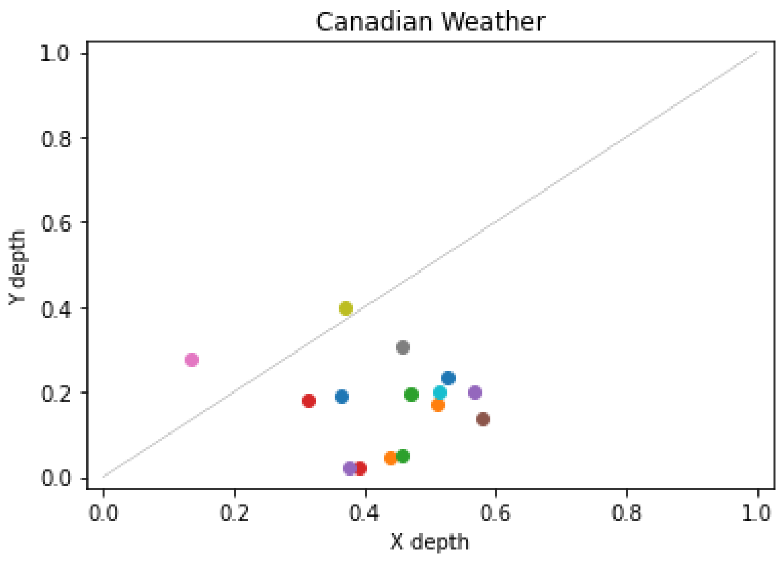
7



```
[12]: dd_plot_mbd2 = DDPlot(fd_temp_atlantic, dist1 = fd_temp_atlantic, dist2 =␣
       ↪fd_temp_continental, depth_method = mbd)

      fig = dd_plot_mbd2.plot()
```

8

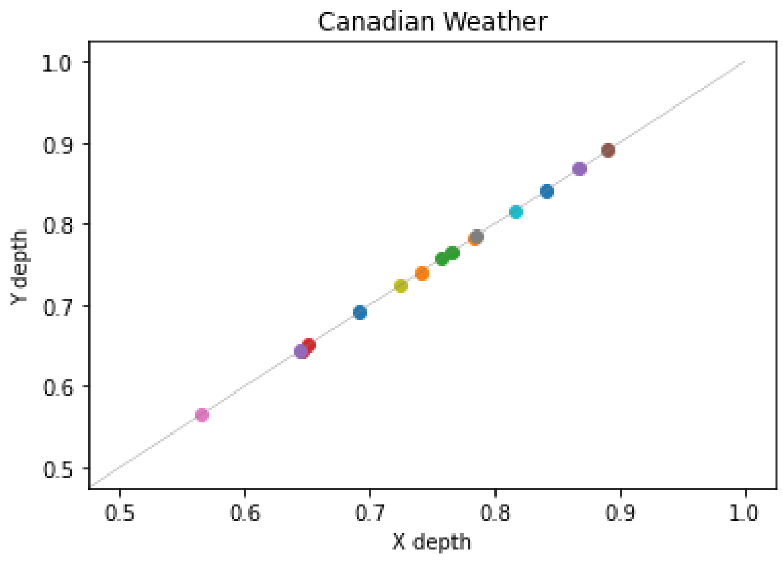


These two graphics compare the atlantic sample with two distributions, both of them are itself. This results on a straight line of points, as the depth of every instance is equal.

[13]:
```
dd_plot_int3 = DDPlot(fd_temp_atlantic, dist1 = fd_temp_atlantic, dist2 =␣
 ↪fd_temp_atlantic, depth_method = int_depth)

fig = dd_plot_int3.plot()
```

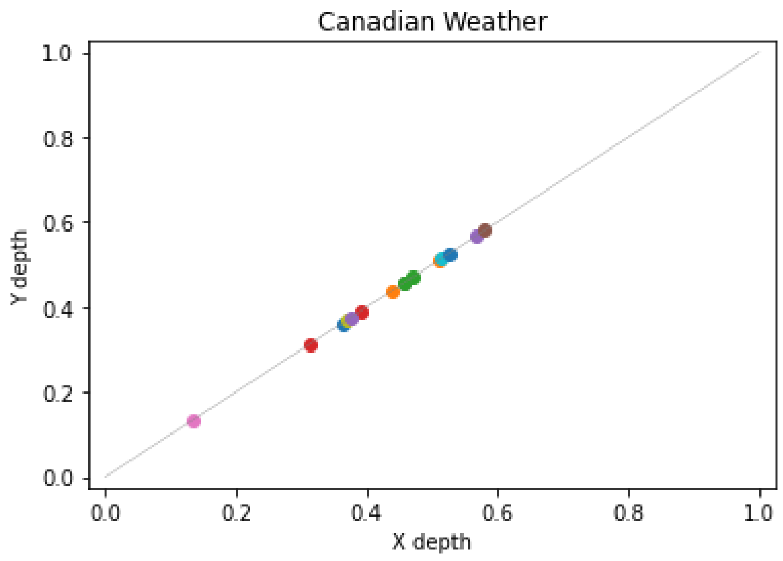
9



```
[14]: dd_plot_mbd3 = DDPlot(fd_temp_atlantic, dist1 = fd_temp_atlantic, dist2 =␣
       ↪fd_temp_atlantic, depth_method = mbd)

      fig = dd_plot_mbd3.plot()
```

10



11

ParametricPlotExamples

May 17, 2021

[1]:
```
# Author: Álvaro Sánchez Romero

from skfda import datasets
from skfda.exploratory.visualization import ParametricPlot
from skfda.representation import FDataGrid

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

In the first example, the parameterization is done with x^3 and its derivative. Both of them are functions with domain 1 and codomain 1, so they can be accepted by the ParametricPlot, that will join them as coordinates to parameterize them.

[2]:
```
x = np.arange(-20,21)
function = lambda x: x ** 3
derivative = lambda x: 3 * (x ** 2)
```

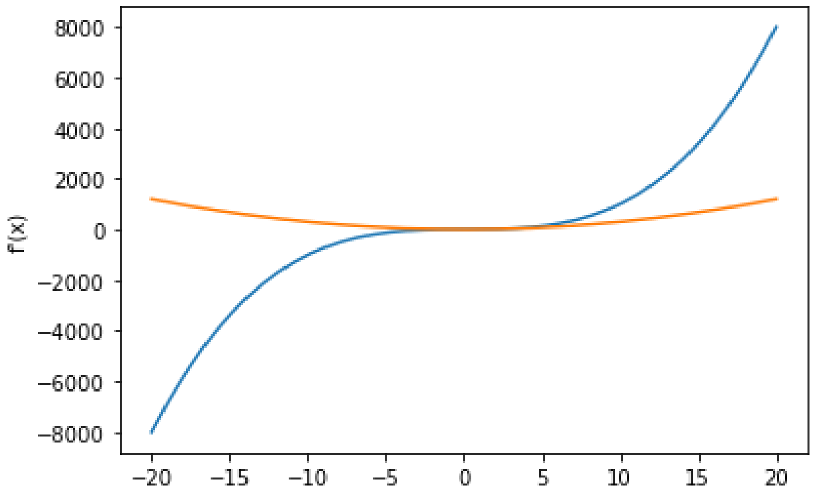
[3]:
```
array_f = np.vectorize(function)
f1 = array_f(x)
array_d = np.vectorize(derivative)
f2 = array_d(x)

fd_func = FDataGrid(f1, x, coordinate_names = ("f(x)",))
fd_der = FDataGrid(f2, x,  coordinate_names = ("f'(x)",))
```

The curves to be parameterized are shown in the next graphic.

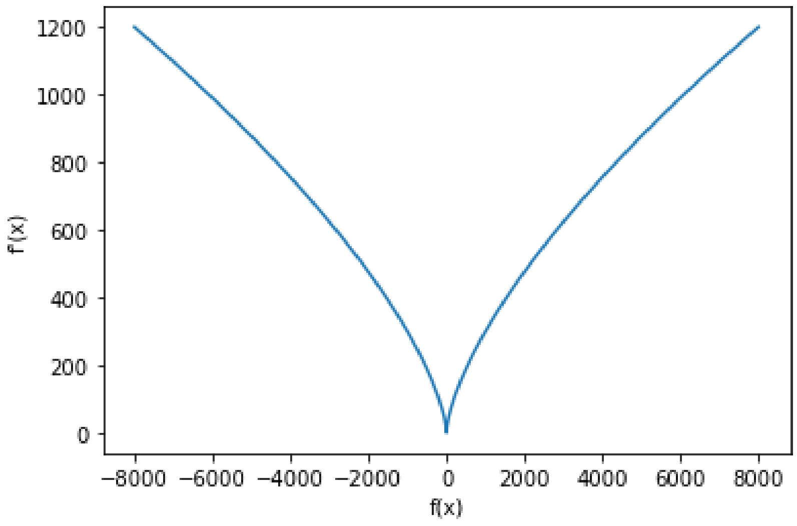
[4]:
```
fig = plt.figure()
fd_func.plot(fig)
fig = fd_der.plot(fig)
```

1



The ParametricPlot gives us the next result when parameterizing the last curves.

[5]:
```
parametric_plot1 = ParametricPlot(fd_func, fd_der)
fig = parametric_plot1.plot()
```



2

In the second example, the parameterization is done with trigonometric functions as sen(x) and cos(x). Both of them are functions with domain 1 and codomain 1, so they can be accepted by the ParametricPlot, that will join them as coordinates to parameterize them.

[6]:
```
x = np.arange(-100,100)
x = x / 8
sen1 = lambda x: math.cos(x)
sen2 = lambda x: math.sin(x)
```

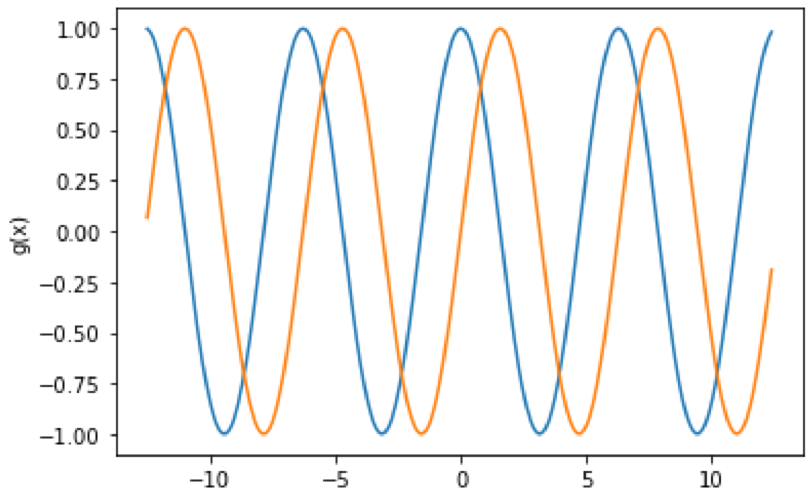
[7]:
```
array_s1 = np.vectorize(sen1)
f3 = array_s1(x)
```

[8]:
```
array_s2 = np.vectorize(sen2)
f4 = array_s2(x)
```

[9]:
```
fd_s1 = FDataGrid(f3, x, coordinate_names = ("f(x)",))
fd_s2 = FDataGrid(f4, x, coordinate_names = ("g(x)",))
```

Their representation can be seen in the next graphic:

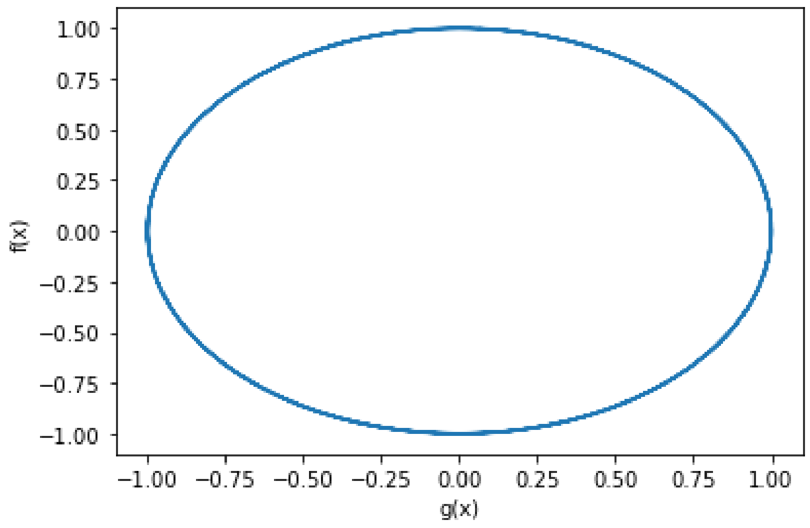
[10]:
```
fig = plt.figure()
fd_s1.plot(fig)
fig = fd_s2.plot(fig)
```



Due to their form the result of their parameterization are loops, in this case a circle.

3

[11]:
```
parametric_plot_sen = ParametricPlot(fd_s2, fd_s1)
fig = parametric_plot_sen.plot()
```

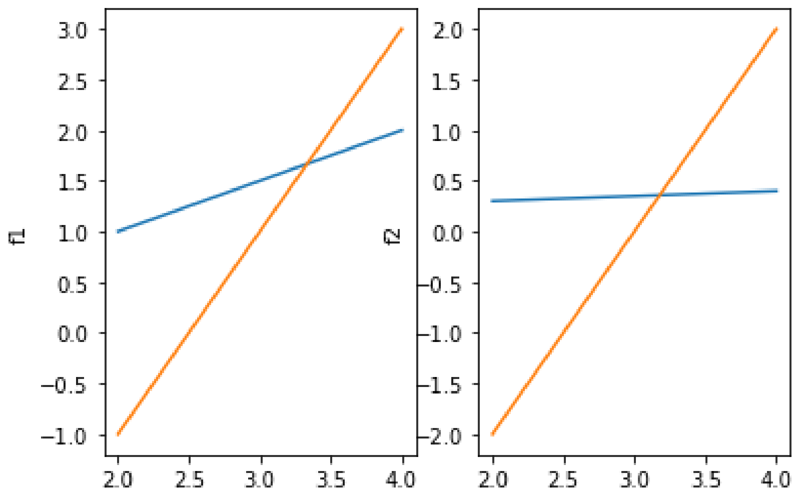


The next example tests uses two curves, a function of codomain of dimension 2.

[12]:
```
data_matrix = [[[1, 0.3], [2, 0.4]], [[-1, -2], [3, 2]]]
grid_points = [2, 4]
fd_comb = FDataGrid(data_matrix, grid_points, coordinate_names = ("f1", "f2"))
```

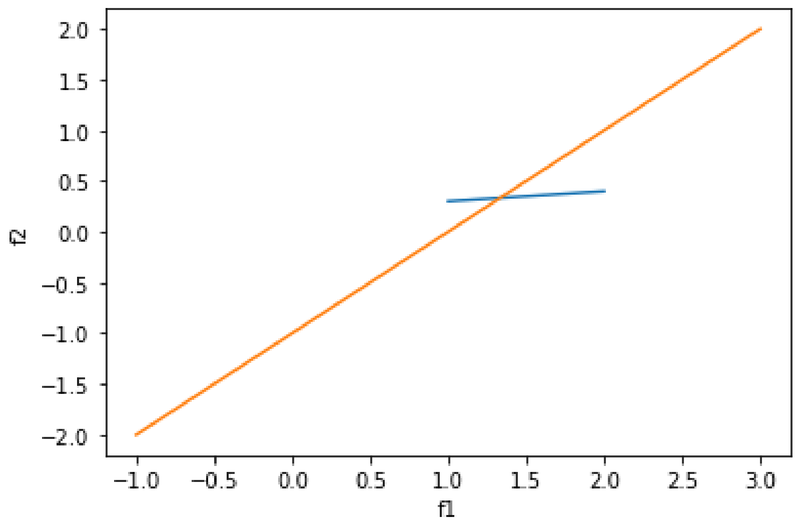
[13]:
```
fig = fd_comb.plot()
```

4



The result of their parameterization can be seen in the next graph:

```
[14]: parametric_plot2 = ParametricPlot(fd_comb)
      fig = parametric_plot2.plot()
```



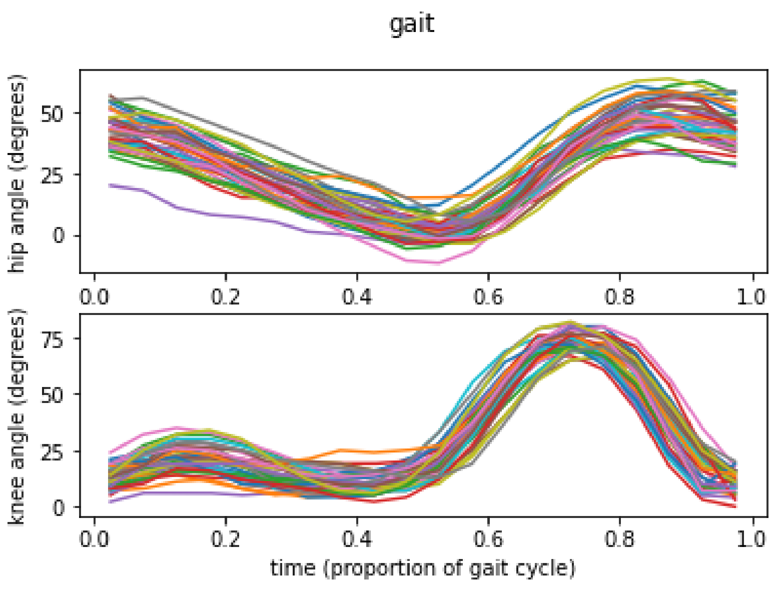
5

In the next example the Gait dataset is loaded ('fda' package from CRAN). This dataset represents the hip angles and knee angles of different children while walking.

[15]:
```
dataset = datasets.fetch_gait()
fd_hip = dataset['data'].coordinates[0]
fd_knee = dataset['data'].coordinates[1]
```

The next graphic has the representation of the gait cycle of the children in the dataset.

[16]:
```
fig, axes = plt.subplots(2)
fd_hip.plot(ax = axes[0])
fig = fd_knee.plot(ax = axes[1])
```



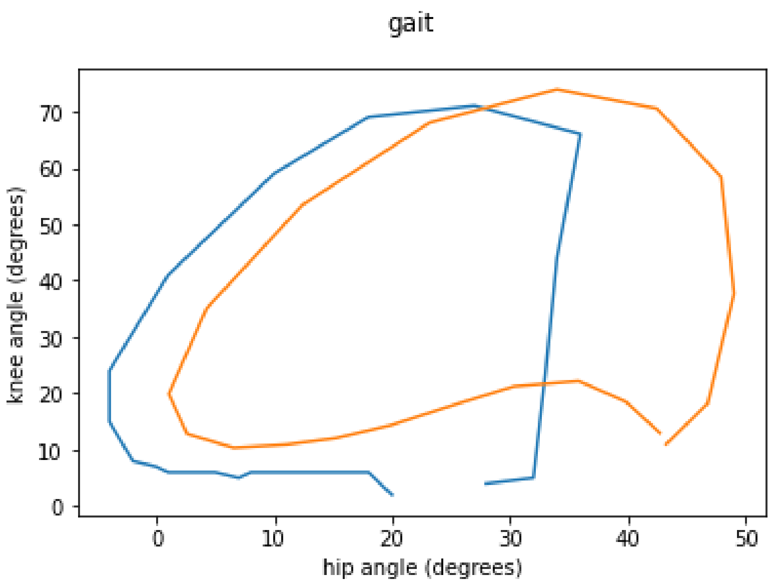
The mean is calculated to compare in a graph the parameterization of a single child hip and knee angles with the one done with the average of all the instances. Thanks to this it will be easier to detect anomalies.

[17]:
```
fd_hip_mean = fd_hip.mean()
fd_knee_mean = fd_knee.mean()
```

This is the result of the parameterization of the mean and one children.

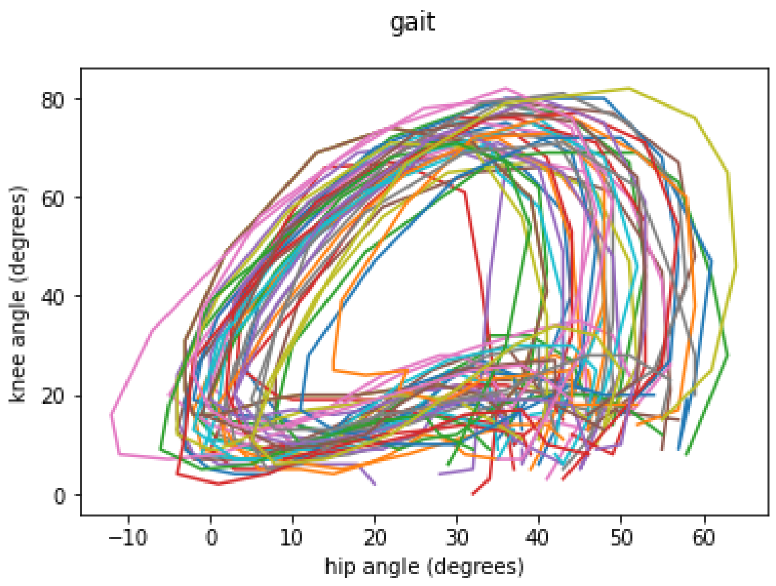
6

```
[18]: parametric_plot_el = ParametricPlot(fd_hip[4], fd_knee[4])
      parametric_plot_el.plot()
      parametric_plot_mean = ParametricPlot(fd_hip_mean, fd_knee_mean, fig =␣
       ↪parametric_plot_el.fig)
      fig = parametric_plot_mean.plot()
```



In the next example it can be seen the parameterization of the whole dataset.

```
[19]: fig = ParametricPlot(fd_hip, fd_knee).plot()
```



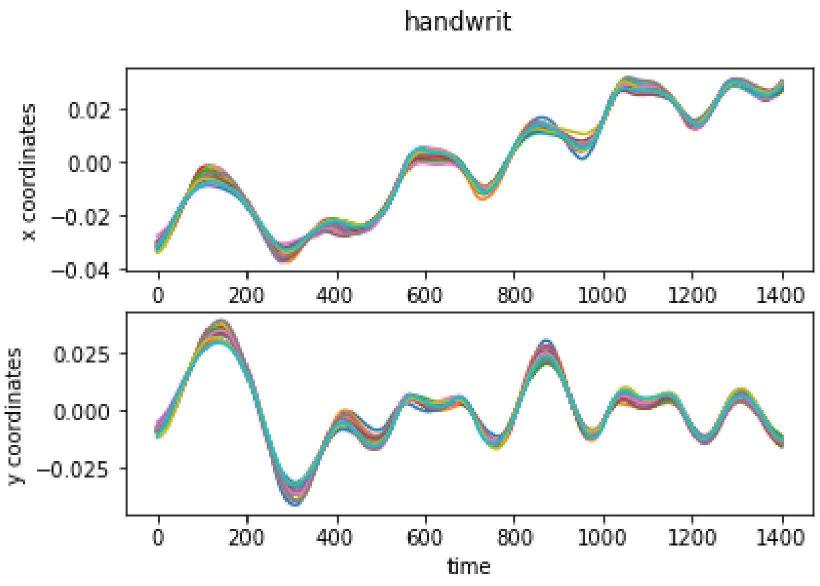
In the next example the Handwrit dataset is loaded ('fda' package from CRAN). This dataset represents the X and Y coordinates obtained while drawing fda.

[20]:
```
dataset = datasets.fetch_handwriting()
```

[21]:
```
fd_x = dataset['data'].coordinates[0]
fd_y = dataset['data'].coordinates[1]
```

[22]:
```
fig, axes = plt.subplots(2)
fd_x.plot(ax = axes[0])
fig = fd_y.plot(ax = axes[1])
```

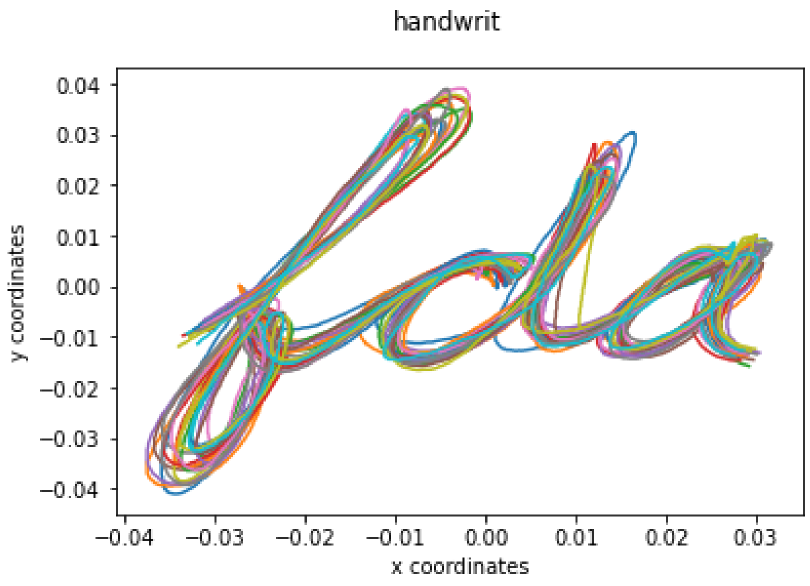
8



As both curves represent the evolution of the coordinates over time, if both of them are parameterized the word fda can be seen.

[23]:
```
fig = ParametricPlot(fd_x, fd_y).plot()
```

9


handwrit
0.04
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
y coordinates
−0.04
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03
x coordinates

10

GraphPlotExamples

May 15, 2021

[1]:
```
# Author: Álvaro Sánchez Romero

# sphinx_gallery_thumbnail_number = 2

from skfda.exploratory.depth import ModifiedBandDepth
from skfda.exploratory.depth import IntegratedDepth
from skfda.exploratory.visualization.representation import GraphPlot
```

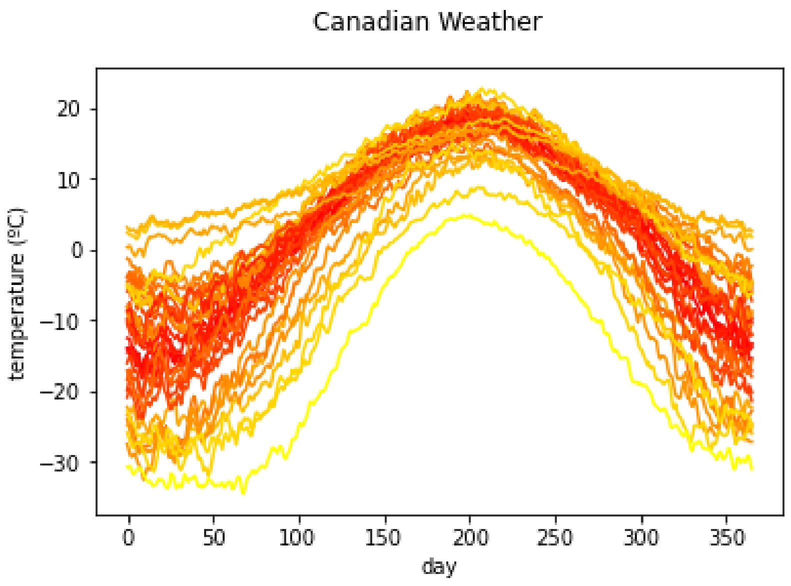
First, the dataset is loaded. In this case, the Canadian Weather dataset from package 'fda' in CRAN is selected. In the experiment, the outliergram cointains the curves representing the temperatures of different weather stations along the year. There are 365 measures, one for each day of the year.

[2]:
```
from skfda import datasets
dataset = datasets.fetch_weather()
fd = dataset["data"]
fd_temperatures = fd.coordinates[0]
```

Thanks to the new feature added, it is possible to represent our data with a gradient of colors that depends on a criteria. This criteria in our example is the computation of the Integrated Depth, allowing the user to see the centrality of the different curves represented.

[3]:
```
depth_in = IntegratedDepth()
graph_integrated_depth = GraphPlot(fd_temperatures, depth_in(fd_temperatures))
fig = graph_integrated_depth.plot()
```

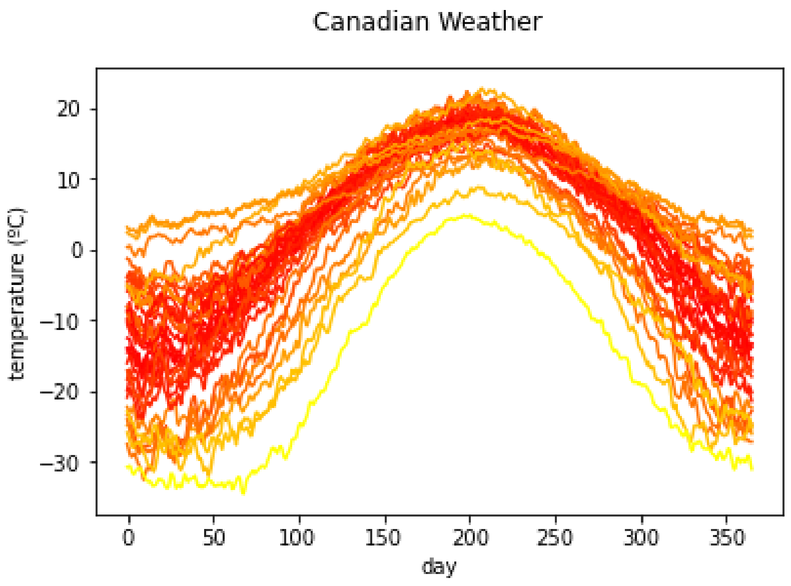
1



Representation of the temperature curves using as criteria the Modified Band Depth.

[4]:
```
mbd = ModifiedBandDepth()
graph_mbd = GraphPlot(fd_temperatures, mbd(fd_temperatures))
fig = graph_mbd.plot()
```

2


Canadian Weather
temperature (ºC)
20
10
0
−10
−20
−30
0
50
100
150
200
250
300
350
day

3