UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Double degree in Computer Science and Mathematics

BACHELOR THESIS

Functional data analysis: interpolation, registration, and nearest neighbors in scikit-fda

Author: Pablo Marcos Manchón
Advisor: Alberto Suárez González

June 2019

Some rights reserved

This work is licensed under a Creative Commons
Attribution-NonCommercial-ShareAlike 3.0 License.
http://creativecommons.org/licenses/by-nc-sa/3.0/
You are free to share (copy, distribute and transmit) and to modify the work
under the following conditions:

- You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- You may not use this work for commercial purposes.
- If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

RIGHTS RESERVED
© June 2019 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, nº 1
Madrid, 28049
Spain

Pablo Marcos Manchón
Functional data analysis: interpolation, registration, and nearest neighbors in scikit-fda

Pablo Marcos Manchón

PRINTED IN SPAIN

The mathematician's patterns,
like the painter's or the poet's must be beautiful;
the ideas like the colours or the words,
must fit together in a harmonious way.

G.H. Hardy, A Mathematician's Apology

Agradecimientos

En primer lugar me gustaría dar las gracias a todas las personas junto a las que he trabajado durante este proyecto. A Alberto, por darme la oportunidad de formar parte del equipo y guiarme a lo largo de este trabajo, dándome libertad para enfocarme en aquellas partes que más me motivaban. A Carlos, por su dedicación y sus exhaustivas revisiones, sin las cuales este proyecto no sería lo que es a día de hoy y a Jose Luis, por sus inestimables aportaciones matemáticas. Finalmente a Manso y Amanda, junto a los cuales he tenido el placer de compartir este trabajo a lo largo del año.

Además me gustaría agradecer el apoyo de mi familia y amigos, el cual ha sido fundamental en esta etapa. En especial a mis padres, sin los cuales nada de esto habría sido posible, a Sergio, por haber estado desde el inicio de los tiempos, y a Lucía, por compartir conmigo este camino y sin la cual todo habría sido muy diferente.

Resumen

El análisis de datos funcionales (FDA) es una rama de la estadística dedicada al estudio de cantidades aleatorias que dependen de un parámetro continuo, como series temporales o curvas en el espacio. En FDA, los datos pueden ser vistos como funciones aleatorias muestreadas de un proceso estocástico subyacente.

Es este trabajo consideraremos tres tareas diferentes en FDA: el uso de técnicas de interpolación para estimar valores de las funciones en puntos no observados, el registro de este tipo de datos y los problemas de clasificación y regresión cuando las instancias son caracterizadas por atributos funcionales. En particular, en este proyecto se han extendido las funcionalidades del paquete de Python *scikit-fda* para dar soporte a estas tres áreas.

En general, las instancias de datos consideradas en FDA están formadas por una colección de observaciones medidas en valores discretos del parámetro del que dependen (p. ej. tiempo o espacio). Para algunas aplicaciones es conveniente, y en algunos casos necesario, estimar el valor de estas funciones en puntos no observados. Esto puede lograrse mediante el uso de interpolación a partir de las mediciones disponibles.

En algunas aplicaciones, las funciones observadas tiene formas similares, pero presentan una variabilidad cuyo origen proviene de distorsiones en la escala del parámetro continuo del que dependen. El registro de los datos consiste en caracterizar esta variabilidad y eliminarla de la muestra.

En este trabajo también se abordan los problemas de clasificación y regresión con datos de naturaleza funcional. Concretamente, se han diseñado estimadores de vecinos próximos, basados en la noción de cercanía entre muestras.

En particular, en este trabajo se ha extendido el paquete *scikit-fda* para incluir métodos de interpolación basados en splines. También se ha dotado el paquete con herramientas para el registro de datos usando traslaciones, puntos de referencia o para el registro elástico, el cual hace uso de la métrica de Fisher-Rao para alinear las funciones de una muestra. Además, se han incluido modelos basados en vecinos próximos para realizar regresión, tanto con respuesta escalar como funcional, y clasificación.

Palabras clave

Análisis de datos funcionales, interpolación, registro, vecinos próximos, python

Abstract

Functional Data Analysis (FDA) is a branch of Statistics devoted to the study of random quantities that depend on a continuous parameter, such as time series or curves in space. In FDA the data instances can be viewed as random functions sampled from an underlying stochastic process.

In this work we consider three different tasks in FDA: the use of interpolation techniques to estimate the values of the functions at unobserved points, the registration of these type of data, and the solution of classification and regression problems in which the instances are characterized by functional attributes. In particular, in this project the *scikit-fda* package for FDA in Python has been extended with functionality in these areas.

Generally, the data instances considered in FDA consist of a collection of observations at a discrete values of the parameter on which they depend (e.g. time or space). For some applications it is convenient, and in some cases necessary, to estimate the value of these functions at unobserved points. This can be achieved through the use of interpolation from the available measurements.

In some applications, the functions observed have similar shapes, but exhibit variability whose origin can be traced to distortions in the scale of the continuous parameter on which the data depend. Registration consists in characterizing this variability and eliminating it from the sample considered.

In this work we also address classification and regression problems with data that are characterized by functions. Specifically, we design nearest neighbors estimators based on the notion of closeness among samples.

Specifically, in this work the *scikit-fda* package has been extended to include interpolation methods based on splines. The package has also been endowed with tools for data registration using either shifts, landmark alignment, or elastic registration, which makes use of the Fisher-Rao metric to align the functions in a sample. In addition, models based on nearest neighbors have been included to carry out regression, with both scalar and functional response, and classification.

Keywords

Functional data analysis, interpolation, registration, nearest neighbors, python

Table of Contents

Lists

List of equations

List of figures

1

Introduction

Functional data analysis (FDA) , is a branch of Statistics that deals with the study of random variables of functional nature, such as time series or curves in the space. It is a relatively recent field, whose first references began in the 1950s, with various articles related to the study of stochastic processes. However, it was not until 1982 with the publication by J.O. Ramsay of *When the data are functions* [1] when the term FDA began to be used to denote this field. Since then, and especially in the last two decades, techniques used for this analysis have evolved quickly, as well as their applications in a wide range of fields such as medicine, bioinformatics or engineering.

Due to their continuous structure, the data treated in FDA are viewed as functions. However, functional data are generally observed and recorded as a discrete set of measures. For this reason, one of the first tasks to be performed in FDA is the representation of these data as continuous functions. One of the main approaches to address this task is the use of interpolation for the construction of continuous functions from these discrete measurements.

Once the data are in functional form, it is possible to perform further analysis, such as studying the variability of a set of samples. Unlike other types of data, due to their functional nature, the variability of a set of functional samples may come from their internal structure, such as the time scale in which a process takes place. For this reason, it is necessary to incorporate a stage in the analysis in which this variability is quantified and separated. This step is called registration.

Throughout this work we will focus primarily on three areas of FDA: the use of interpolation for data representation, the registration, and the use of nearest neighbors estimators on classification and regression problems with functional data.

There are some software solutions in R and Matlab that provide support to this field, such as *fda* [2] [3]; or *fda.usc* [4], developed by a team at the University of Santiago de Compostela. Inspired by these solutions, in 2017, the *scikit-fda* project arose under the name *fda* [5], as part of a Bachelor's degree thesis made by Miguel Carbajo at the UAM. The aim of this work was to initiate the creation of a Python package to give support to this field, under the philosophy of an open-source project, along with the creation of a community that contributes to its development and maintenance.

Currently, the project is being driven by the Machine Learning Group at the UAM together with the contributions of several undergraduate thesis, such as this one. The package is already in use by some researchers and was presented in the *III International Workshop on Advances in Functional Data Analysis* [6].

1.1. Goals and scope

The aim of this work is to extend the functionality provided by the *scikit-fda* software package. The functionalities that will be developed during this work may be divided into three main areas.

Firstly, interpolation techniques will be integrated with the existing structures for the representation of functional data. These original structures do not allow the evaluation of functional data as if they were functions, which will be possible after the incorporation of the contributions of this work.

Secondly, tools will be created to the data registration. For this task the books *Functional data analysis* [7] and *Functional and shape data analysis* [8] have been taken from reference. As a result of these contributions, the package will provide an API to perform this step of the analysis.

To conclude, classification and regression models based on nearest neighbours will be incorporated. For this purpose, the estimators of the *scikit-learn* [9] API will be used as a reference, incorporating an extension of these estimators for functional data into the *scikit-fda* package.

1.2. Document Structure

The main part of the present document is divided into three chapters, aside from this introduction part. In the Chapter 2, it is exposed the state of the art, which gives an overview of the mathematical framework of the functionalities developed to the package.

Chapter 3 summarizes the analysis and design of the functionalities added to the package, as well as the methodology and technologies used during the development of the work. To finish the main part, Chapter 4 presents the conclusions of the work done along with a brief discussion about future work.

Besides, this document appends three annexes. In Annex A there are more detailed descriptions of the different algorithms used, along with proofs of some mathematical results set out in Section 2.3, which have been moved to the annexes for the sake of clarity.

Finally, Annex B includes a series of Python notebooks showing examples of the use of the functionalities incorporated, which have been thought to show in a pleasant way how to use the package to a new user. The documentation of the software developed has been include in the Annex C.

2

State of the art: interpolation, registration, and nearest neighbors

In FDA, the analyzed data can take the form of temporal series, curves, surfaces or any process that varies over a continuum. The observations that make up our data come from random variables, whose realizations take values in functional spaces, that is to say, they are realizations of stochastic processes. For this reason, they can be viewed as random functions.

In practice, the information of a functional datum $f(t)$ is observed and recorded as pairs (t_j, y_j), where y_j is a snapshot of the function f at t_j, i.e., $f(t_j)$. One may wonder what makes FDA different from multivariate analysis, considering the fact that data is generally recorded and stored in the form of observations at discrete points [8]. In multivariate statistics one works with the vector of values $\{y_j\}$, applying statistical methods and vector calculus to its study, but without taking into account the high correlation between the values y_j, and y_{j+1}, due to the continuous structure given by the parameter on which they vary. However, FDA keeps the association of values y_j and t_j, taking into account the functional nature of the data, allowing the study of the data using functional calculus. Since the observations are functions, dependent on one or several continuous parameters, such as the time in which a process takes place, it is not possible to measure and store all the points which they are defined for.

There are two main approaches in FDA to represent the data while preserving its functional structure. The first, called *discrete representation*, stores in a finite grid the pairs (t_j, y_j) which representes the values $f(t_j)$. Interpolation is used to reconstruct the data as a continuous function from its discrete values.

The second one is a parametric approach, called *basis representation*, in which is considered a system of functions $\{\phi_k(t)\}_{k=1}^K$, such as a truncated Fourier basis or a set of polynomials. This system forms a finite subspace of the original functional space, where the observations are projected, obtaining a representation associated to the coefficients of the basis $f(t) \approx \sum_{k=1}^K c_k \phi_k(t)$.

Due to this functional structure in the data, FDA needs to address some issues that do not arise when studying other types of data, such as vectors. For instance, once our data is represented as a set of functions $\{f_i(t)\}_{i=1}^n$, its variability can proceed from two sources, the random curve-to-curve variation, i.e., the difference between the values of the functions, or from the internal structure of its domain. It is fundamental to study and quantify these two types of variability in the analysis of our data.

This separation is called the registration of the data, and is treated in detail in the Section 2.2.

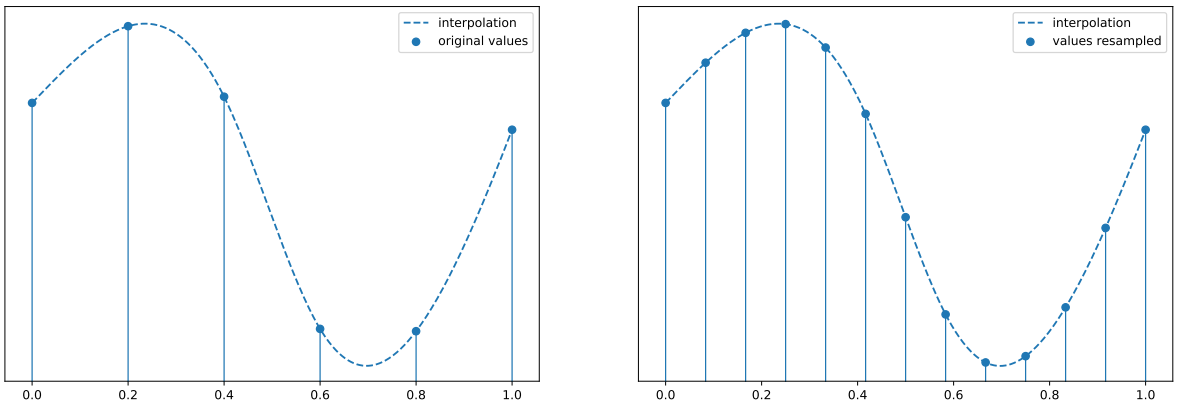
FDA also deals with topics present in classical statistics in cases where the data are functions. Among these problems are the study of regression and classification models. In the Section 2.4 the use of nearest neighbors estimators applied to functional data is studied.

Although in FDA is also treated the case in which our data are multivariate functions, from $\mathbb{R}^d$ to $\mathbb{R}^m$, in general, as a way of simplification, during this chapter we will assume that our data are univariate functions.

In this chapter is made an overview, from a theoretical point of view, of different FDA topics covered in this work. However, this chapter also serves to introduce the functionalities included in the *scikit-fda* package. The figures shown throughout this section have been made using these functionalities.

2.1. Interpolation

In the discrete representation, frequently, it is necessary to resample or evaluate the data, or its derivatives, at points within the domain range, different that the original pairs $\{(t_j, y_j)\}$ at which our observations have been measured. An example of this it is shown in the Figure 2.1. For this purpose interpolation is used. This allows to treat each functional datum as a single entity $f(t)$, which varies continuously throughout t.



(a) Original observation

(b) Observation resampled

Figure 2.1: Function resampled using interpolation

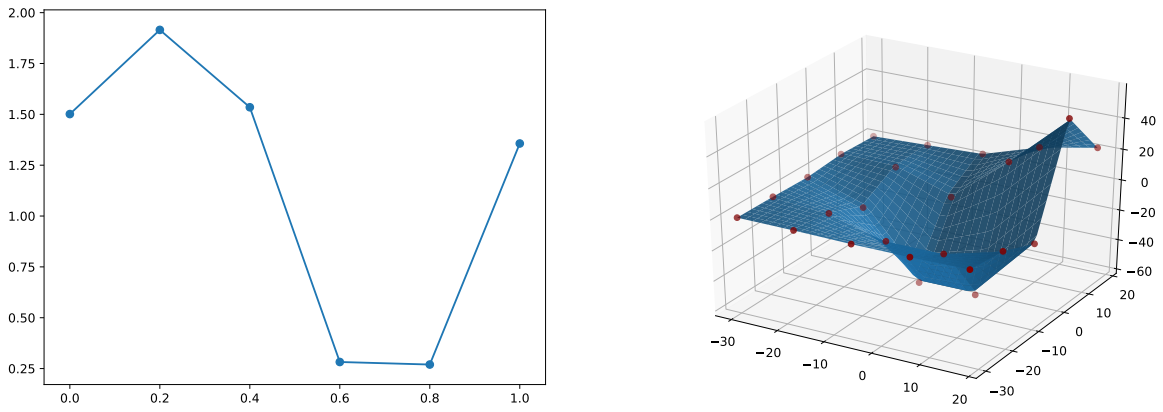
Although they are not the only methods used for interpolation, splines and smoothing splines are the most used, for this reason we will focus on them during this work.

2.1.1. Linear interpolation

Given two points t_k and t_{k+1} for which the values of our function are known, denoted by y_k and y_{k+1}, we can use a line segment to join these values as a first approach. Using this interpolation, we will obtain a piecewise linear function, whose values in the interval $[t_k, t_{k+1}]$ are be given by

$$f(t) = y_k + \frac{y_{k+1} - y_k}{t_{k+1} - t_k}(t - t_k)\,. \tag{2.1}$$

In the case of multivariate functions, multilinear interpolation generalizes this method to higher dimensions. The same principles are applied, but the piecewise domain is composed by regions calculated using triangulation, and the evaluation is performed using a multilinear form, obtaining in the case of surfaces piecewise functions formed by plane sections. In the Figure 2.2 it is plotted the result of interpolating a temporal series and a surface.



(a) Linear interpolation

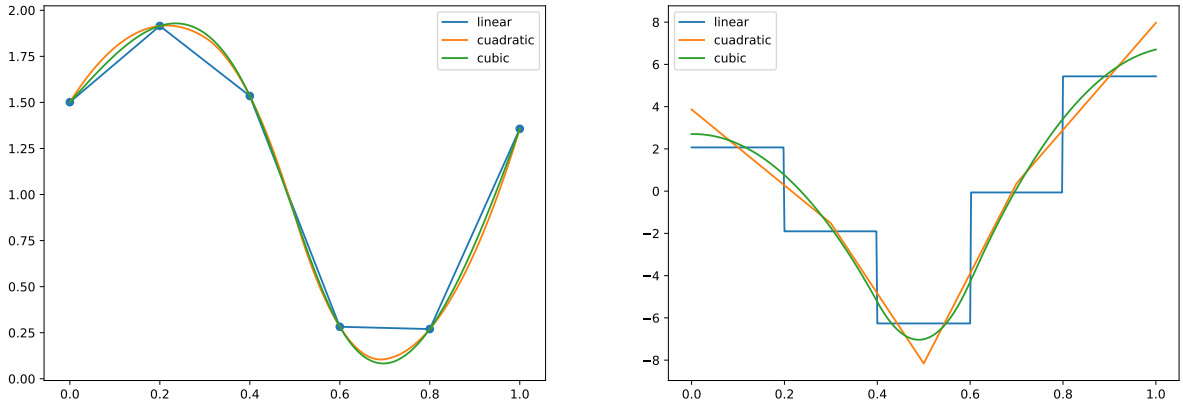
(b) Bilinear interpolation

Figure 2.2: Example of linear interpolation

2.1.2. Spline interpolation

In spline interpolation [10], the points are linked by a piecewise polynomial. The main advantage of using splines of order p, i.e., using polynomials of order p in the different intervals, is that we will not only obtain continuous functions, as in the linear case, but we will also obtain functions $\mathcal{C}^{p-1}[a, b]$, i.e., that are continuous and have $p - 1$ continuous derivatives. This fact is crucial in FDA, because we will need to use derivatives in many parts of the analysis.

To achieve this, we have to match the values of the derivatives of the adjacent splines in the interpolation knots. If we denote by $s_k(t) = \sum_{j=0}^{n} c_{jk} t^k$ to the spline defined in the region $[t_{k-1}, t_k]$, during the calculation of the coefficients c_{jk}, we must impose the restriction $s_k^{(d)}(t_k) = s_{k+1}^{(d)}$ for

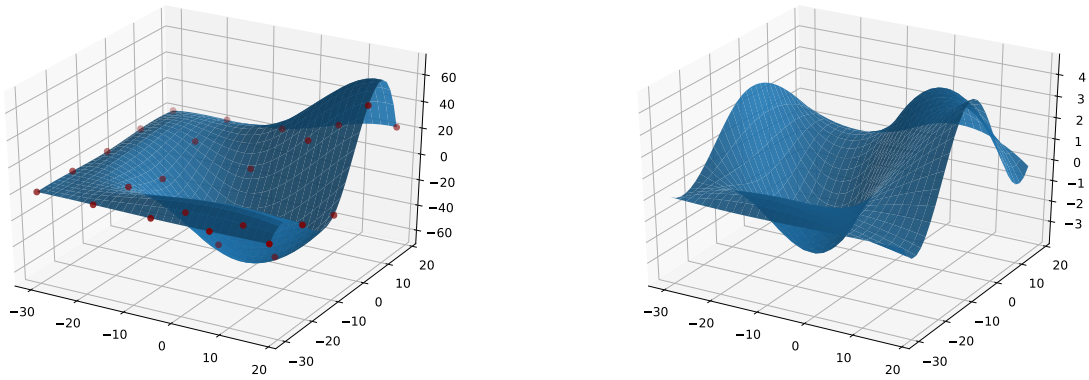


(a) Spline interpolation

(b) First derivatives

Figure 2.3: Example of spline interpolation

$d = 1, \ldots, p-1$. For this purpose, we will define a linear system of equations which can be solved iteratively. Figure 2.3(a) shows the result of interpolate a temporal series using splines of different orders, and the first derivatives of these splines, which are splines of order $p - 1$ due to the derivation of the polynomials.



(a) Surface interpolated

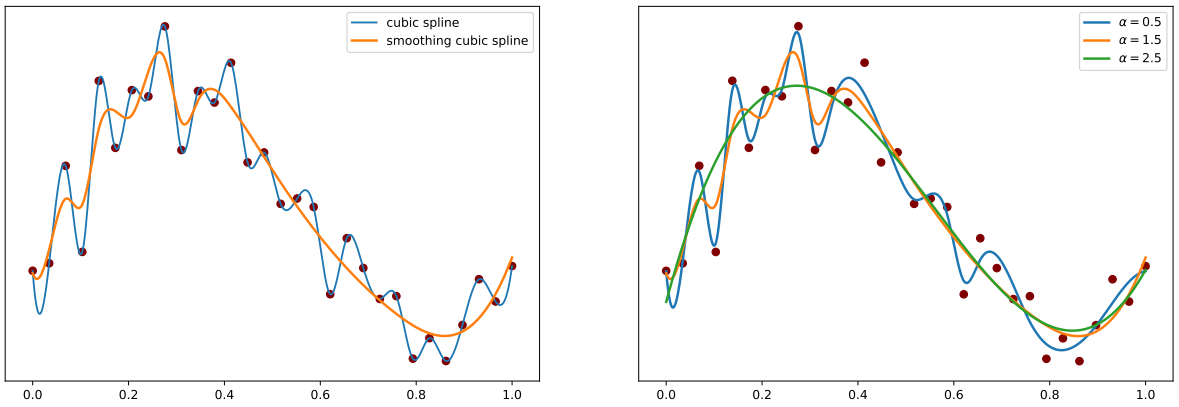
(b) Partial derivative ∂_x

Figure 2.4: Interpolation of surface with bicubic splines

We can extend spline interpolation for multivariate functions, such as surfaces, where bivariate splines will be used. In this case, triangulation is used to calculate the regions in which the polynomials are defined. These polynomials are of the form $s(x, y) = \sum_{0 \leq i+j \leq n} c_{i,j} x^i y^j$. In the Figure 2.4(a) it is shown the result of the interpolation of a surface using bicubic splines and the partial derivative respect to its first parameter.

2.1.3. Smoothing spline interpolation

In several contexts, the functional data are contamined with noise. This contamination can be due to errors in the measure procedure or to the intrinsic random behaviour of the process, this will motivate the use of smoothing splines.



(a) Cubic smoothing

(b) Several values of the smoothing factor

Figure 2.5: Smoothing interpolation

This method is a variant of the spline interpolation, wich not uses directly the points $\{t_k\}_{k=1}^N$ as knots. Instead, it is used a smaller set of knots $\{\tilde{t}_k\}_{k=1}^M$, generally different from the originals, with their corresponding values $\tilde{y}_k = f(\tilde{t}_k)$ calculated using spline interpolation. The function interpolated using smoothing splines is defined using spline interpolation on the set of knots $(\tilde{t}_k, \tilde{y}_k)$. Figure 2.5 shows the smoothing interpolation of a noisy observation.

To determine the knots $\tilde{t}_k$ is used an smoothing parameter α. A weight w_k is assigned to each of the original points t_k, used to calculate the knots $\tilde{t}_k$ as a weighted average of the original ones. These weights can be set uniformly. The number of knows will be increased until the smoothing condition is satisfied, defined as

$$\sum_{k=1}^{N} \left(w_k \left(y_k - \tilde{f}(t_k) \right) \right)^2 \leq \alpha \tag{2.2}$$

where $\tilde{f}(t_k)$ is the value of the smoothing spline at t_k. In other words, the smoothing parameter sets the maximum value of the residuals.

After representing the data as continuous functions, we can move on to other parts of the analysis, such as the study of variability due to the continuous structure of the data, called registration of the data. The techniques used during the registration of the data require the use of a discrete representation of the functions and their evaluation in arbitrary points, which is done by interpolation.

2.2. Registration

In many situations, functional observations have similar shapes, but they are in some way misaligned. This variability can interfere with further analysis. Therefore, it is desirable quantify and eliminate or reduced this variation. This process is called *registration*.

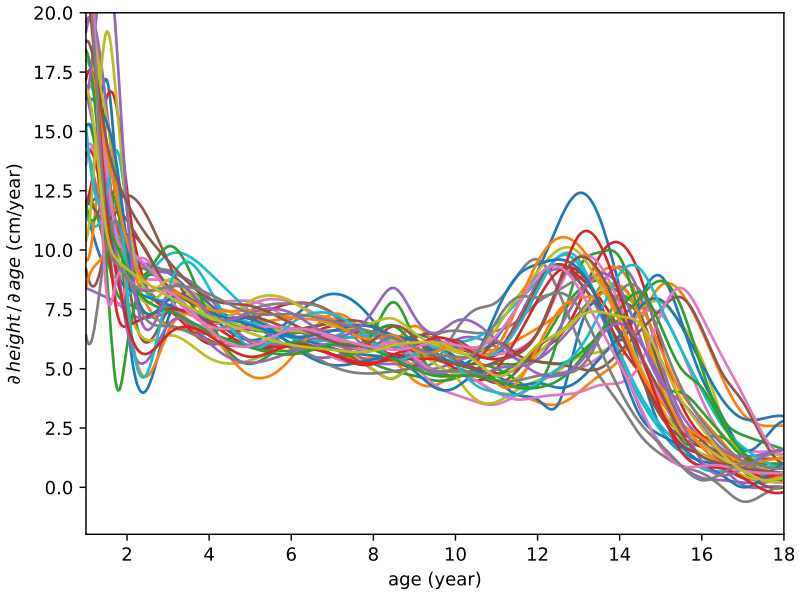
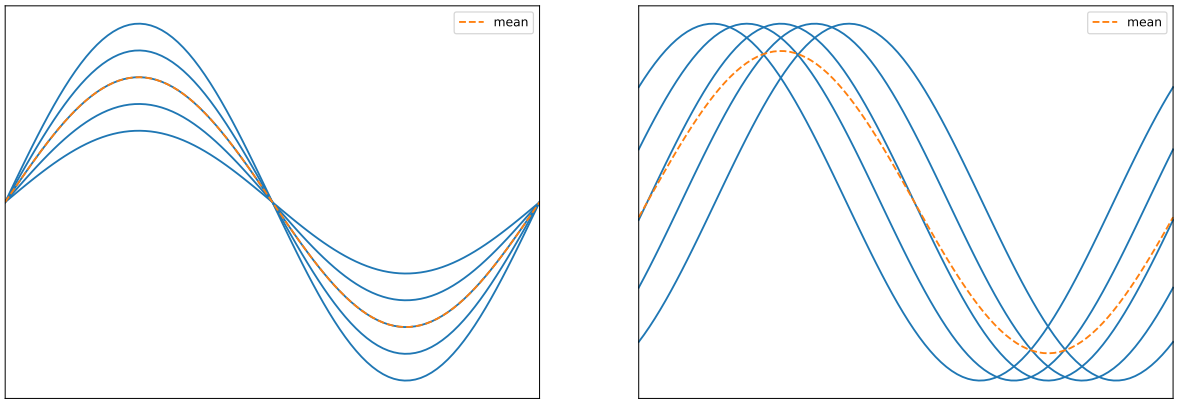


Figure 2.6: Male growth rate

To illustrate the problem we consider functional data from the Berkeley Growth Study [11]. In this 1954 study, the heights of 54 girls and 39 boys between the ages of 1 and 18 years were recorded. In Figure 2.6 velocity growth curves of the boys are shown. These curves correspond to the derivatives of the growth curves. In them this type of variability may be appreciated in a more pronounced way. Every curve shows a main stage of growth corresponding with the puberty; however, these stages are not aligned due to hormonal and other physiological factors in the growth. The rigid metric of physical time may not be directly relevant to the internal dynamics of our problem. Thus it may be convenient to make a transformation of the time scale to adapt it to the nature of our data. In the registration of the data, or alignment, this type of variability is analyzed, quantified and separated.

The variability of the data can be analyzed in terms of amplitude and phase variation. *Amplitude variation* corresponds to the random curve-to-curve variation. In the Figure 2.7(a) a set of curves whose variability proceeds exclusively from the amplitude is shown. *Phase variation* refers to misalignments of the curves with respect to its domain. In the Figure 2.7(b) a set of curves whose variation is completely due to the phase is shown; this source of variability is the one that will be dealt with in the registration process.



(a) Dataset with amplitude variability

(b) Dataset with phase variability

Figure 2.7: Amplitude and phase variability

2.2.1. Shift registration

A first approach to solve this problem was presented in Ramsay and Silverman (2005) [7], by considering a simple shift in the domain to make the registration. Although it is a basic approach, it will be useful in many cases. Figure 2.8(a) shows a set of sinusoidal waves whose phases are not aligned, and a shift in the time scale will be adequate to make the alignment.

Let $\{f_i\}_{i=1}^n$ be a set of functional observations, which will be aligned using this transformation. We are actually interested in finding the values

$$f_i^*(t) = f_i(t + \delta_i) \qquad i = 1, 2, \dots n \tag{2.3}$$

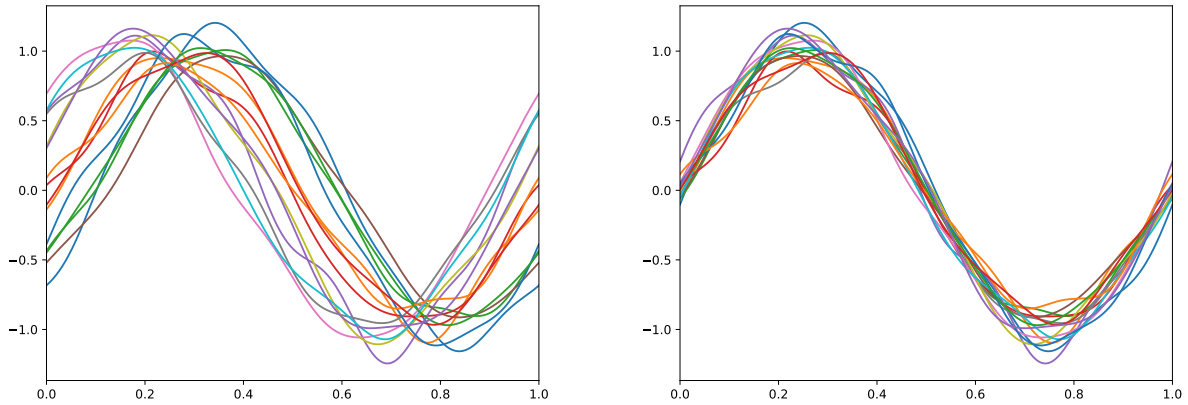
where the shift parameter δ_i is chosen in order to align the features of the curves. In the Figure 2.8(b) it is shown the result of apply this type of transformation to the set of sinusoidal waves.

A possible solution for the calculation of shifts δ_i, is using a least square criterion, minimizing the Registered sum of squared errors (REGSSE) , defined as

$$\text{REGSSE} = \sum_{i=1}^{n} \int_{\mathcal{T}} [f_i(t + \delta_i) - \hat{\mu}(t)]^2 \, dt \tag{2.4}$$

where $\hat{\mu}(t)$ is the mean of the registered functions $f_i(t + \delta_i)$.

The alignment problem will be based on finding the values δ_i that minimize this sum of errors. For this purpose, we will use the derivatives of the functions x_i, using a variant of the Newton-Rhapson method. A description of the algorithm used to calculate these values is given in the Appendix A.1.



(a) Unregistered curves $x(t)$

(b) Registered curves $x^*(t)$

Figure 2.8: Shift registration of a dataset

2.2.2. Warping functions

In most cases, it is necessary to consider a more general transformation than a simple translation; generally, this transformation will not be linear. In the case of univariate functional data, i.e., the case that the functions under analysis depend on a single continuous parameter, we will have functions $f_i : \mathcal{T} \subset \mathbb{R} \to \mathbb{R}$, in such a way that we can understand the problem as the search for an appropriate parameterization of our data, according with the intrinsic structure of the dataset.

We will consider the functions $\gamma_i : \mathcal{T} \to \mathcal{T}$, referred to as warping functions in the related literature, that we will use to reparametrize the domain, i.e, to change the internal scale of the data. Using these warping functions we will obtain the curves registered by means of composition of functions, i.e.,

$$f_i^*(t) = f_i(\gamma_i(t)) = f_i \circ \gamma_i. \tag{2.5}$$

So that the alignment does not alter the structure of the functional data, these functions γ_i must be boundary-preserving dipheomorphisms. A dipheomorphisms is an invertible function that maps one differentiable manifold to another such that both the function and its inverse are smooth. In our case the dipheomorphisms maps the domain of the functions f_i, $\mathcal{T}$, to itself. The boundary-preserving condition imposes that the border of $\mathcal{T}$ is not altered by the warping functions. In the case where the domain of the functions $\mathcal{T}$ is an interval $[a, b]$, the warpings will be strictly increasing functions that fix the bounds of the domain, i.e., $\gamma_i(a) = a$ and $\gamma_i(b) = b$, as could be seen in the Figure 2.9.

Without loss of generality, in the following sections, we will assume that $\mathcal{T} = [0, 1]$, because the general case $\mathcal{T} = [a, b]$ can be reduced to this with an affine transformation. Also, we will denote the set of such warping functions as Γ.

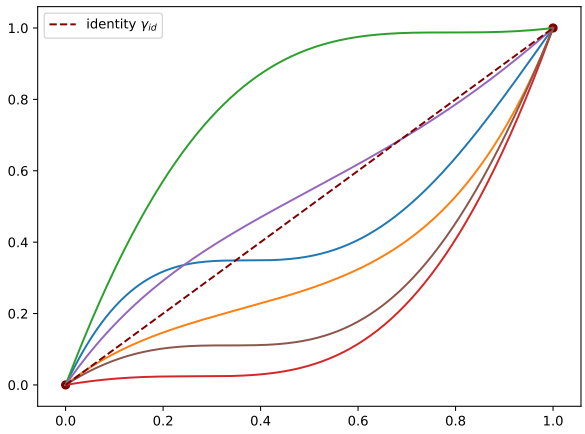
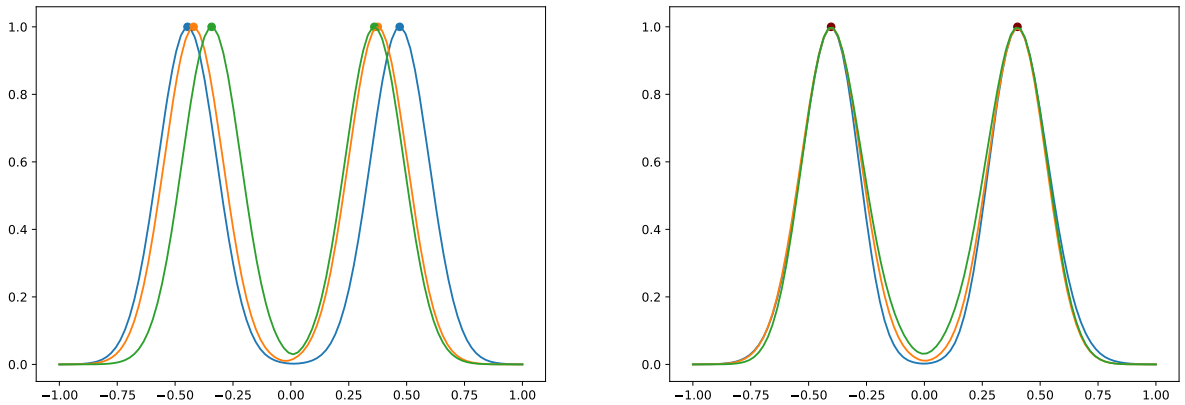


Figure 2.9: Set of warping functions defined in $\mathcal{T} = [0, 1]$

2.2.3. Landmark registration

A possible solution to register a dataset is to select a set of points for each of the observations and align these points using a warping function. This method is called landmark registration.

A landmark, or feature of a curve, is some characteristic that can be associated with a specific point of the domain. There are typically maximums, minimums or zero crossings points. For instance, the population shown in Figure 2.10 has two distinctive features, formed by the maximum points of each of the samples.



(a) Unregistered dataset $f_i(t)$

(b) Registered population $f_i^*(t)$

Figure 2.10: Shift registration of a dataset

The landmark registration process will require, for each observation f_i, the identification of the values $\{t_{ij}\}_{j=1}^{F}$ associated with each of the F features, which will be aligned to a common point $\{t_j^*\}_{j=1}^{F}$. Once we have the landmarks points, we will build the warping functions so that $\gamma_i(t_j^*) = t_{ij}$, thus

$$f_i^*(t_j^*) = f_i(\gamma_i(t_j^*)) = f_i(t_{ij}). \tag{2.6}$$

Not all sets of populations have differentiated characteristics of this type, moreover, by taking into account only a limited number of points the resulting alignments can become quite artificial, altering the internal structure of the samples. This will motivate us to build methods which use a global criterion for alignment and not just a discrete number of points.

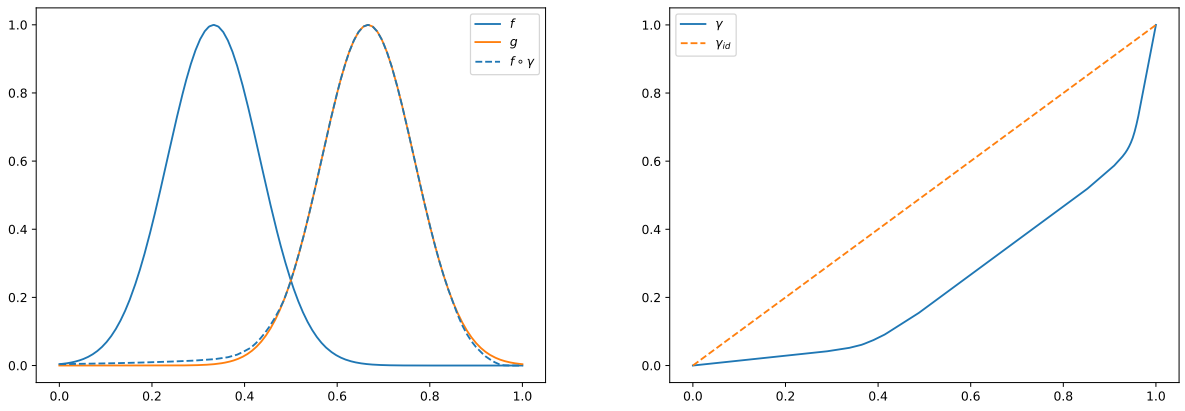
2.2.4. Pairwise and groupwise alignment

We will define the pairwise alignment problem [8] to deal with the registration problem of two functions using a global criterion.

Let $f_1, f_2 \in \mathcal{F}$ be functional observations of a general space $\mathcal{F}$ and $E : \mathcal{F} \times \mathcal{F} \to \mathbb{R}^+$ an energy functional. The alignment problem may be understood as the search of a warping function γ^* which minimizes the energy between the two functions, i.e.,

$$\gamma^* = \underset{\gamma \in \Gamma}{\operatorname{argmin}}\, E[f_1, f_2 \circ \gamma]. \tag{2.7}$$

When a warping γ^* fulfills this property, we will say that f_1 is registered to $f_2 \circ \gamma^*$. In the Figure 2.11(a) it is shown an example where two functions have been registered using this approach.



(a) Alignment of f_1 to f_2

(b) Warping built by the DPA algorithm

Figure 2.11: Pairwise alignment

To estimate γ^*, we will explore different paths that the reparameterization may take in a discretized grid, trying to minimize the energy term. This algorithm, described in deatil in [8], is called Dinamic programming algorithm (DPA) , because it makes use of dynamic programming techniques to search

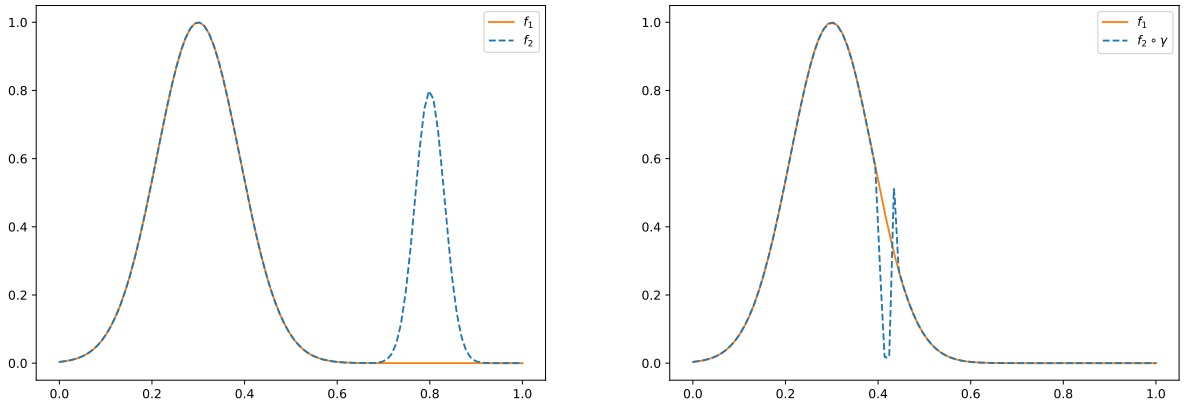
the optimal path [12].

Given a set of functions $\{f_i\}_{i=1}^n \subset \mathcal{F}$, the groupwise alignment problem will consist in the search of warping functions $\{\gamma_i^*\}_{i=1}^n \subset \Gamma$ to align each of the functions with the rest of them. To achieve this, we will build a target function μ, also called template, to which all the curves will be aligned. For instance, the cross-sectional mean of the functions $\bar{f} = \frac{1}{n}\sum_{i=1}^n f_i$ can be used as target.

A possible choice for the energy term would be to take $E[f_1, f_2] = \|f_1 - f_2\|^2_{\mathbb{L}^2} = \int_{\mathcal{T}}(f_1 - f_2)^2$. This criterion is not commonly used in practice because three problems will arise that will not make it adequate: the pinching effect, the lack of symmetry and the inverse inconsistency [13]. In what follows we provide a a short description of these issues.

Pinching effect

The pinching effect can be observed when two functions without a perfect match are aligned, taking the $\mathbb{L}^2$ distance as energy. To minimize the energy term E in eqn. 2.7, the optimal solution will tend to squeeze a region whose features make it difficult to align, until it disappears. Figure 2.12 shows the pinching effect on the registration of two functions. A part of f_2 is identical to f_1 over [0, 0.6] and completely different over the remaining domain. The optimal solution will tend to squeeze the second part of f_2, until it vanishes completely.



(a) f_1 and f_2

(b) Pinching effect

Figure 2.12: Pinching force effect

Lack of symmetry

Let $f_1, f_2 \in \mathcal{F}$ be two functions that are aligned following the pairwise criterion (eqn. 2.7). Generally, if we apply a reparameterization $\gamma \in \Gamma$ to both functions, their distance will change,

$$\|f_1 \circ \gamma - f_2 \circ \gamma\|^2 = \int_{\mathcal{T}} (f_1(\gamma(t)) - f_2(\gamma(t)))^2 dt = \int_{\mathcal{T}} |f_1(s) - f_2(s)|^2 \frac{1}{\dot{\gamma}(\gamma^{-1}(s))} ds \qquad (s = \gamma(t)), \quad (2.8)$$

in other words, the action of Γ over $\mathcal{F}$ under the $\mathbb{L}^2$ metric is not by isometries.

Therefore $E[f_1, f_2] \neq E[f_1 \circ \gamma, f_2 \circ \gamma]$ and consequently $f_1 \circ \gamma$ y $f_2 \circ \gamma$ may not be aligned. A direct consequence of this issue will be the inverse inconsistency of the problem, because of $E[f_1 \circ \gamma, f_2] \neq E[f_1, f_2 \circ \gamma^{-1}]$. It would be desirable that the transformation necessary to align f_1 to f_2 be the inverse transformation to align f_2 to f_1. The search of a more adequate energy for this problem will motivate the Section 2.3 of this work.

2.2.5. Amplitude phase decomposition

It is useful to quantify the amount of variability due to the phase and the amplitude, in order to understand the data and validate the registration procedure. For this purpose, in Kneip and Ramsay (2008) [14] it is developed an effective method for quantifying this variation once the data have been aligned.

Let $\{f_i\}_{i=1}^N$ be a set of functional observations, and $\{f_i^*\}_{i=1}^N = \{f_i \circ \gamma_i\}_{i=1}^N$ the corresponding registered samples, and $\bar{f}$, $\bar{f}^*$ the cross-sectional means. The Total Mean Square Error is defined as

$$\text{MSE}_{total} = \frac{1}{N}\sum_{i=1}^{N} \|f_i(t) - \overline{f}(t)\|^2 = \frac{1}{N}\sum_{i=1}^{N} \int [f_i(t) - \overline{f}(t)]^2 dt. \tag{2.9}$$

This error can be decomposed as $\text{MSE}_{total} = \text{MSE}_{amp} + \text{MSE}_{phase}$. This decomposition allow separate the variability due to the phase and the amplitude.

$$\text{MSE}_{amp} = C_R \frac{1}{N}\sum_{i=1}^{N} \int \left[f_i^*(t) - \overline{f^*}(t)\right]^2 dt \quad \text{MSE}_{phase} = \int \left[C_R \overline{f^*}^2(t) - \overline{f}^2(t)\right] dt \tag{2.10}$$

Where C_R is a constant related with the covariance between $\dot{\gamma}_i$ and $(f_i^*)^2$. From this decomposition it is possible to define the square error multiple correlation index, which indicates the proportion of the variation due to the phase explained by the registration process. This index is defined as

$$R^2 = \frac{\text{MSE}_{phase}}{\text{MSE}_{total}}. \tag{2.11}$$

For instance, by quantifying the alignment of the dataset of the Figure 2.8, we get a value of $R^2 = 0{,}99$, which indicates that the $99\,\%$ of the variability it is produced by the phase.

2.3. Elastic methods in functional data analysis

As discussed in previous chapter, the variability in functional data may come from the continuous structure of its domain. This problem does not appear exclusively in functional data analysis, other fields such as shape analysis or computer vision should deal with this kind of variability.

In the classical approach, phase variation is separated in the registration of the data, as a preprocessing step. However, in the elastic analysis approach the separation of the two sources of variability is incorporated as a fundamental part of the analysis. The term *elastic* comes from the fact that we will allow deformations of the domain of the functions throughout the analysis.

In Srivastava et. al. (2011) present in [15] a novel framework to treat this approach using the Fisher-Rao metric. Although this framework covers a wide variety of topics, such as classification, regression or functional component analysis [16], in this chapter we will focus on the part concerning the so-called elastic registration of functional data. For that, we will follow the explanation outlined on chapters 1, 3, 4, and 8 of the book Functional and Shape Data Analysis [8] and the implementation in the R-package *fdasrvf* [17].

2.3.1. Fisher-Rao metric

As introduced in Section 2.2.4, the use of the metric $\mathbb{L}^2$ for the pairwise alignment of functions leads to a series of difficulties, which make it unsuitable for this problem. This is why we will use differential geometry techniques to find a suitable metric. In particular, given two functions f_1, f_2 and $\gamma \in \Gamma$, we look for a metric that remains invariant to warpings in the domain, i. e., $d(f_1, f_2) = d(f_1 \circ \gamma, f_2 \circ \gamma)$.

For this purpose we will use the Fisher-Rao metric. This Riemannian metric was introduced in 1945 by C. R. Fisher in a version on the space of probability distributions. In our case we will use a non-parametric version, slightly different from the original one, defined on the space of signed measures. This metric plays a very important role in information geometry, and it is the only metric that possesses this property of invariance to warpings in the domain [18].

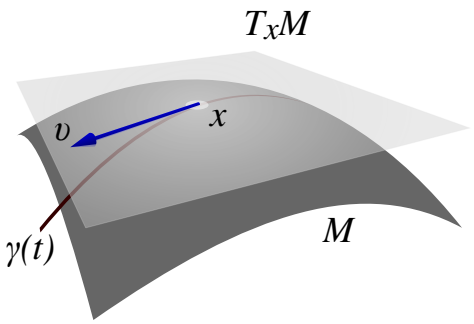


Figure 2.13: Tangent space of tridimensional manifold M [1]

A Riemannian metric is an application that assigns an inner product at each point of variety on its tangent space that varies smoothly from point to point. This will give us local notions of angles, length of curves or volumes.

The tangent space of a manifold facilitates the generalization of vectors to general manifolds, since one cannot simply subtract two points to obtain a vector that gives the displacement of the one point from the other. The tangent space of a point x in a general manifold M, denoted as T_xM, is made up of the velocities at x of all the paths of the variety passing through the point. For example, in the Figure 2.13 is shown a manifold M in $\mathbb{R}^3$. This velocities of the path are tangent vectors to the curve. In this case the tangent space T_xM made up of all these vectors is a plane.

In our case the manifold will be formed by the set of absolutely continuous functions $\mathcal{F}$, that is, those functions whose derivative exists a.e. and belongs to the space of square integrable functions $\mathbb{L}^2$. And the path between points of $\mathcal{F}$ may be understood as smooth deformations of the functions. We will endow $\mathcal{F}$ with the Fisher-Rao metric.

Let $f \in \mathcal{F}$ and $v_1, v_2 \in T_f(\mathcal{F})$, the Fisher-Rao metric is defined as

$$\langle\langle v_1, v_2 \rangle\rangle_f = \frac{1}{4} \int_0^1 \dot{v}_1(t) \dot{v}_2(t) \frac{1}{|\dot{f}(t)|} dt, \tag{2.12}$$

where $\dot{v}$ denotes the derivative of v.

Using this local notion of inner product, we will be able to calculate the length of $\alpha(\tau) \subset \mathcal{F}$, a differentiable path in our manifold as

$$L[\alpha] = \int_0^1 \langle\langle \dot{\alpha}(\tau), \dot{\alpha}(\tau) \rangle\rangle_{\alpha(\tau)} d\tau. \tag{2.13}$$

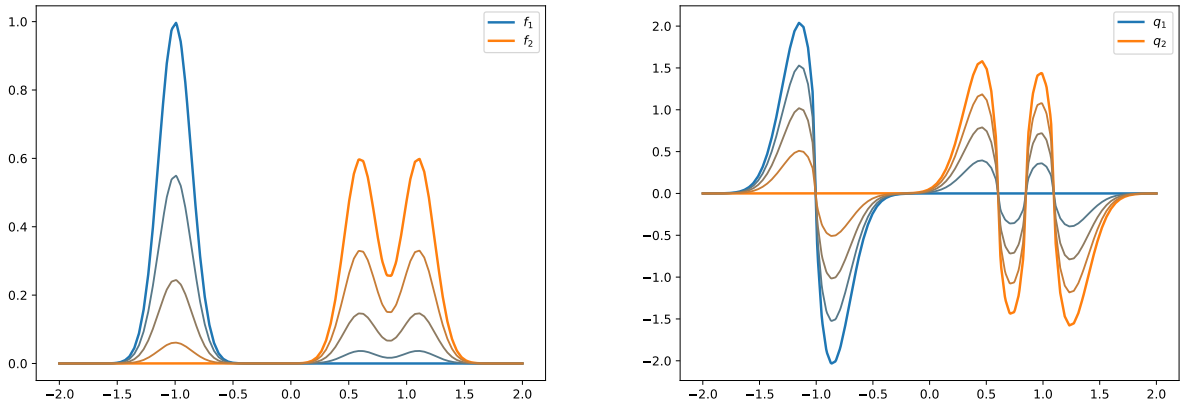
This allows us to define the distance between two points of the manifold $f_1, f_2 \in \mathcal{F}$ as the length of the geodesic path, or shortest path, which connects f_1 and f_2,

$$d_{FR}(f_1, f_2) = \inf_{\alpha:[0,1]\to\mathcal{F}, \alpha(0)=f_1, \alpha(1)=f_2} L[\alpha]. \tag{2.14}$$

This path is the shortest smooth deformation between f_1 and f_2. Figure 2.14(a) illustrates some steps in the deformation between f_1 and f_2. Finding this geodesic path directly is computationally unapproachable, for this reason we will introduce the Square Root Slope Funcion (SRSF) transform, which simplifies this computation.

Given $f \in \mathcal{F}$, its SRSF is defined as $SRSF\{f\} = \operatorname{sgn}(\dot{f})\sqrt{|\dot{f}|}$. To simplify notation, in the subsequent sections, we will denote the SRSF of a function $f_i \in \mathcal{F}$ as q_i.

[1] Derivative work: McSush. Original uploader: TN at German Wikipedia. [Public domain]



(a) Five steps of the geodesic path between f_1 and f_2

(b) SRSF of the path

Figure 2.14: Geodesic path in $\mathcal{F}$ and the corresponding SRSF's

Under this transformation, the Fisher-Rao metric becomes the usual metric in $\mathbb{L}^2$, so that the distance between two functions will be calculated using the distance $\mathbb{L}^2$ of their corresponding SRSF's, i.e., $d_{FR}(f_1, f_2) = \|q_1 - q_2\|_{\mathbb{L}^2}$. A proof of this result is included in appendix A.2.

Taking advantage of this characterization we will take our functions to the SRSFs space to perform the analysis efficiently, and then transform the result to the original space. This can be done because the SRSF defines a map up to constant between $\mathcal{F}$ and the space of SRSFs with the $\mathbb{L}^2$ metric. A consequence is that the computation of geodesics shown in 2.14 becomes in a straight line between SRSFs, $\alpha(\tau) = (1-\tau)q_1 + \tau q_2 \quad 0 \leq \tau \leq 1$.

Given $\gamma \in \Gamma$, the SRSF of $f \circ \gamma$ is

$$SRSF\{f \circ \gamma\} = \text{sgn}(\dot{f} \circ \gamma)\sqrt{|\dot{f} \circ \gamma|}\sqrt{\dot{\gamma}} = (q \circ \gamma)\sqrt{\dot{\gamma}}. \tag{2.15}$$

To simplify the notation, the SRSF of this composition is denoted by (q, γ). Using this fact, we can proof that the action of Γ on $\mathcal{F}$ is an action by isometries, i.e, given $f \in \mathcal{F}$ and $\gamma \in \Gamma$ the composition $f \circ \gamma$ preserves the metric. A proof of this result is given in the appendix A.2. Figure 2.15 shows a diagram with the action of the composition in the different spaces.

$$\begin{array}{ccc} f & \xrightarrow{\text{SRSF}} & q \\ \text{action on } \mathcal{F} \downarrow & & \downarrow \text{action on } \mathbb{L}^2 \\ f \circ \gamma & \xrightarrow{\text{SRSF}} & (q, \gamma) \end{array}$$

Figure 2.15: Action of Γ

2.3.2. Amplitude and phase spaces

Due to the properties of the Fisher-Rao metric, we can formally decompose the distance between two functions of $\mathcal{F}$ into the part due to phase and amplitude.

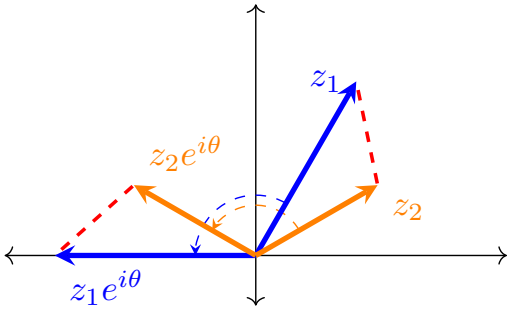
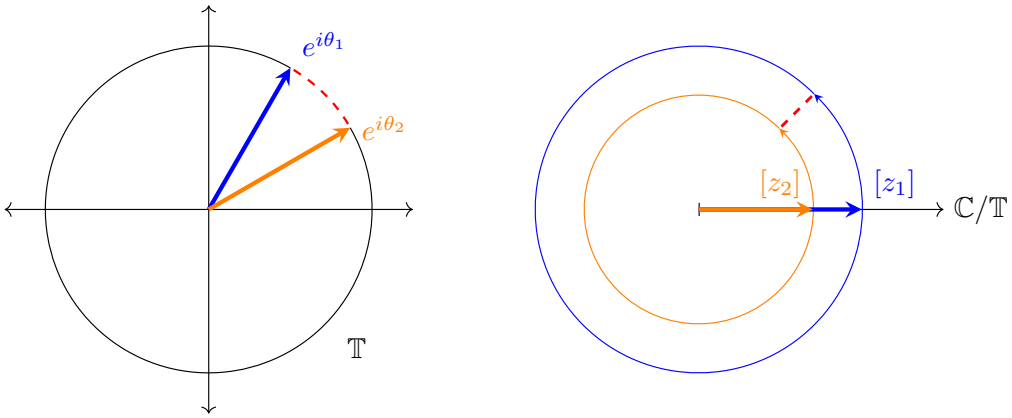


Figure 2.16: Rotation in complex plane

Firstly, it is useful to consider and analogy with the rotations on the complex plane, to understand the role by the phase and the amplitude in the manifold $\mathcal{F}$.

Let z_1, z_2 be two points in $\mathbb{C}$, as it is illustrated in the Figure 2.16, when a rotation on the plane is applied the distance between z_1 and z_2 remains invariant, i.e., it is an action by isometries, as the reparameterizations on our manifold.

The variability between two vectors in the plane is completely specified by the angle between them, which will play the role of the phase in our space, and the difference of their modules, which will be equivalent to the distance between the vector once aligned.



(a) Angle between vectors

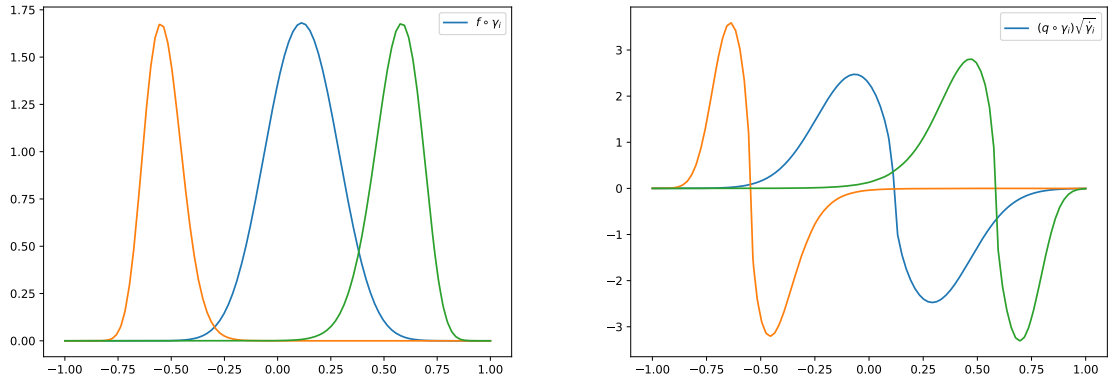
(b) Distance in $\mathbb{C}/\mathbb{T}$

Figure 2.17: Phase and amplitude in the complex plane

This distance of the vectors aligned may be understood as a distance in the quotient space $\mathbb{C}/\mathbb{T}$, where $\mathbb{T}$ is the unit circle, which it is isomorph to the group of rotations in the plane $SO(2)$. In this

quotient space we will define the equivalence classes $[z] = \{ze^{i\theta} : \theta \in [0, 2\pi]\}$.

In the case of our manifold, let $q \in \mathbb{L}^2$ be a SRSF. We will define its orbit under Γ as $[q] = \{(q, \gamma) : \gamma \in \Gamma\}$, in other words, it is the set of reparameterizations associated to q. In the Figure 2.18(a) it is shown some reparameterizations associated to a function of $\mathcal{F}$ and their corresponding SRSF's in 2.18(b).



(a) $f \circ \gamma_i$ (b) (q, γ_i)

Figure 2.18: Functions in the same orbit

We will denote amplitude space $\mathscr{A} = \mathbb{L}^2/\Gamma$ to the set of these orbit. In this space the phase variation is incorporated within equivalence classes, while the amplitude variation appears across equivalence classes, as in the analogy with the complex plane. We will endow the space with the elastic metric, defined as

$$d_a\left([q_1], [q_2]\right) = \inf_{\gamma_1, \gamma_2 \in \Gamma} \left(\|(q_1, \gamma_1) - (q_2, \gamma_2)\|\right), \tag{2.16}$$

To quantify the other source of variability, we will define the phase space, which will denoted as $\Gamma = \{\gamma : [0, 1] \to [0, 1] : \gamma$ is a boundary-preserving diffeomorphism$\}$, for which the natural distance will be given by the Fisher-Rao metric.

Let $\gamma \in \Gamma$ be a warping function. Under the Fisher-Rao metric, the norm of such γ is 1, thus

$$\|\gamma\|_\Gamma^2 = \|SRSF(\gamma)\|_{\mathbb{L}^2}^2 = \|\dot{\gamma}\|_{\mathbb{L}^2}^2 = \int_0^1 \sqrt{\dot{\gamma}(t)}\sqrt{\dot{\gamma}(t)}dt = \int_0^1 \dot{\gamma}(t)dt = \gamma(1) - \gamma(0) = 1. \tag{2.17}$$

The SRSF transform is an isometry between Γ and the unit sphere in $\mathbb{L}^2$, also known as Hilbert Sphere $\mathbb{S}_\infty = \{\psi \in \mathbb{L}^2 : \|\psi\|_{\mathbb{L}^2} = 1\}$. To calculate distances in Γ we will apply this transformation which will allow us to take advantage of the structure of $\mathbb{S}_\infty$.

Let γ_1, γ_2 be in Γ and $\psi_1 = \sqrt{\dot{\gamma}_1}, \psi_2 = \sqrt{\dot{\gamma}_1}$ be their SRSF, their distance in Γ will be given by

$$d_{phase}(\gamma_1, \gamma_2) = d_\psi(\psi_1, \psi_2) = \cos^{-1}\left(\int_0^1 \psi_1(t)\psi_2(t)dt\right). \tag{2.18}$$

This result is analogous to the rotations in the plane, where the cosine of the angle between two unit vectors will be given by the inner product. If we apply a rotation to two vectors, their angle remains invariant, in our case the phase distance will be invariant to common reparameterizations due to the properties of the Fisher-Rao metric.

Let f_1, f_2 be functions in $\mathcal{F}$, their relative phase is defined as

$$(\gamma_1^*, \gamma_2^*) = \underset{\gamma_1,\gamma_2\in\Gamma}{\operatorname{argmin}} \|(q_1, \gamma_1) - (q_2, \gamma_2)\| \in \Gamma \times \Gamma. \tag{2.19}$$

Thus allow us to calculate the phase variability between them, as a generalization of the notion of an angle. The phase distance between these functions is the distance of their relative phase $d_{phase}(\gamma_1^*, \gamma_2^*)$. Alternatively, the properties of the metric can be used to define the relative phase depending on a single warping, $\gamma_{12}^* = \underset{\gamma\in\Gamma}{\operatorname{argmin}} \|(q_1, \gamma) - q_2\|$, which is quivalent to set γ_2^* to the identity in 2.19.

2.3.3. Pairwise alignment

The Fisher-Rao metric allow us to formulate a suitable criterion for the pairwise alignment problem, as presented in Section 2.2.4, that avoids all the problems associated with the metric $\mathbb{L}^2$.

Given $f_1, f_2 \in \mathcal{F}$, to register f_1 to f_2 the Fisher-Rao distance will be minimized as energy term E (see eqn. 2.7), i.e., the warping used in the alignment will be

$$\gamma^* = \operatorname{argmin}_{\gamma\in\Gamma} d_{FR}(f_1 \circ \gamma, f_2) = \operatorname{argmin}_{\gamma\in\Gamma} \|(q_1, \gamma) - q_2\|_{\mathbb{L}^2}. \tag{2.20}$$

The energy used to align the functions in Figure 2.11(a) is actually this quantity. Since $E[f_1, f_2] = E[f_1 \circ \gamma, f_2 \circ \gamma]$, the warping used to register $f_1 \circ \gamma$ to $f_2 \circ \gamma$ will be the same as in the previous case. Because of this property, our problem will have inverse consistency, since

$$E[f_1 \circ \gamma, f_2] = E[f_1 \circ \gamma \circ \gamma^{-1}, f_2 \circ \gamma^{-1}] = E[f_1, f_2 \circ \gamma^{-1}], \tag{2.21}$$

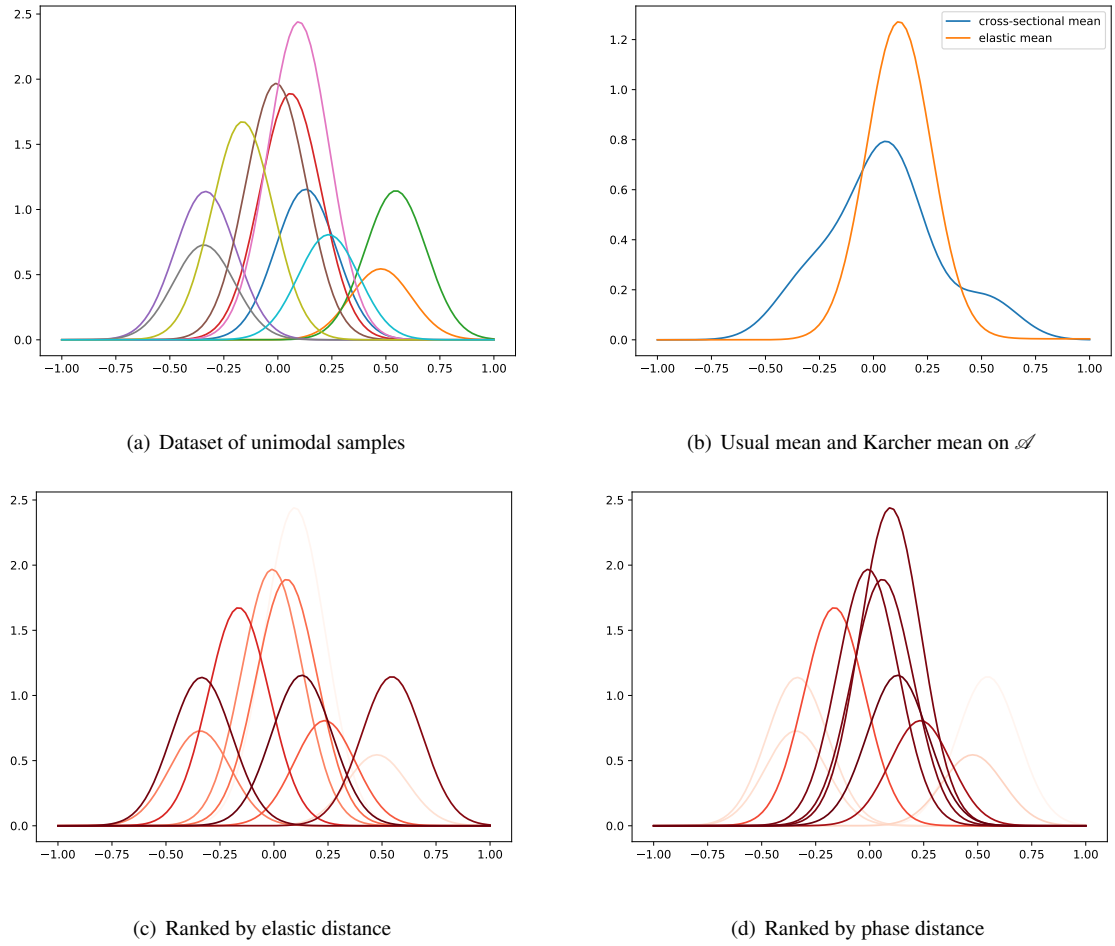
so that the warping needed to register f_2 to f_1 is ${\gamma^*}^{-1}$. In addition, the so-called pinching effect will not appear, due to the term $\sqrt{\dot{\gamma}}$ that is present in the criterion to minimize.

2.3.4. Karcher means

In descriptive statistics, given a set of random points $\{x_i\}_{i=1}^N \subset \mathbb{R}^n$, the sample mean $\bar{x} = \frac{1}{N}\sum_{i=1}^N x_i$ is used to estimate the central tendency of the data. The mean is the quantity that minimizes the sum of square distances.

In the functional case, the mean function will have this property thus if $\{f_i\}_{i=1}^N \subset \mathbb{L}^2$ is a set of functional observations, their mean $\bar{f}(t) = \frac{1}{N}\sum_{i=1}^N f_i(t)$ will minimize $\sum_{i=1}^N \|f_i - \bar{f}\|^2_{\mathbb{L}^2}$.

We are interested in extend this idea to general metrics spaces. Let (X, d) be a metric space and $\{x_i\}_{i=1}^N$ random points in X. The Fréchet variance of a point $p \in X$ is defined as $\Psi(p) = \sum_{i=1}^N d^2(p, x_i)$.



(a) Dataset of unimodal samples

(b) Usual mean and Karcher mean on $\mathscr{A}$

(c) Ranked by elastic distance

(d) Ranked by phase distance

Figure 2.19: Karcher mean of dataset

The Fréchet mean of these random points will be defined as the element $m \in X$ which globally minimizes the Fréchet Variance. If this global minimum does not exist, we will call Karcher means to the points that locally minimizes the variance $\Psi(p)$, i. e.,

$$m = \underset{p \in M}{\arg\min} \sum_{i=1}^{N} d^2\left(p, x_i\right) \tag{2.22}$$

These Karcher means are adapted to the metrics used in the analysis. For this reason they can better capture the geometry of our problem than usual mean, as it is shown in the Figure 2.19(b), where it is used a Karcher mean on $\mathscr{A}$. The dataset shown in 2.19(a) contains Gaussian-like samples, however the usual mean of Figure 2.19(b) is not able to reflect this shape, unlike the Karcher mean in $\mathscr{A}$.

As an example of the behavior of these means with different metrics, the centrality of an observation in a dataset can be measured using its distance to the Karcher mean. In the Figure 2.19 it is used this idea to rank a dataset of unimodal samples. Reddish colors indicate higher centrality of a sample, i.e., a smaller distance to the mean, and lighter colors indicates outlier samples. In the Figure 2.19(c) it is used the amplitude distance, and their corresponding Karcher mean. We can observe that the location of the mode in a sample does not affect its centrality, in contrast to the phase distance, as it is shown in the Figure 2.19(d).

2.3.5. Elastic registration

In this section we will define a procedure, which will be called elastic registration, to perform a groupwise alignment of a dataset under this framework. A target function will be created, called the elastic mean, to which all samples will be aligned later, as discussed in Section 2.2.4.

Let $\{f_i\} \subset \mathcal{F}$ be a dataset of functions to register and q_i their corresponding SRSFs. Firstly we will compute their Karcher mean in $\mathscr{A}$, defined as

$$[\mu_q] = \underset{[q]\in\mathscr{A}}{\operatorname{arginf}} \sum_{i=1}^{n} d_a\left([q],[q_i]\right)^2. \tag{2.23}$$

The bracket notation $[\mu_q]$ is used to emphasize the fact that it is an orbit in a quotient space. We will need a criterion to select a particular element of this orbit, for that reason we will define the center of the orbit as the element $\tilde{\mu}_q \in [\mu_q]$ such that the relatives phases γ_i^* between q_i and $[\mu_q]$ have as Karcher mean the identity.

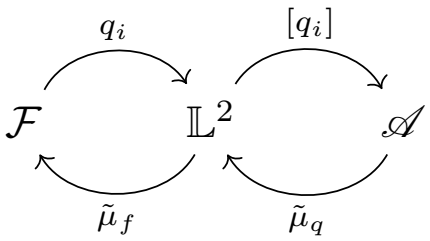


Figure 2.20: Scheme of the elastic registration procedure.

We will select an arbitrary element μ_q of $[\mu_q]$ to which we will calculate the corresponding warping functions to align the set of SRSFs, i.e., $\gamma_i = \operatorname{arginf}_{\gamma\in\Gamma} \|\tilde{q} - (q_i, \gamma)\|$. Then, we will compute the Karcher mean of $\{\gamma_i\}$, denoted as $\bar{\gamma}$, in the phase space. The center of the orbit will be $\tilde{\mu}_q = (\mu_q, \bar{\gamma}^{-1})$. The mean of the relative phases with respect to the center of the orbit $\tilde{\mu}_q$, will be the identity, thus

$$\operatorname{arginf}_{\gamma\in\Gamma} d_{FR}(\gamma_i \circ \bar{\gamma}^{-1}, \gamma) = \operatorname{arginf}_{\gamma\in\Gamma} d_{FR}(\gamma_i \circ \bar{\gamma}^{-1} \circ \bar{\gamma}, \gamma \circ \bar{\gamma}) = \operatorname{arginf}_{\gamma\in\Gamma} d_{FR}(\gamma_i, \gamma \circ \bar{\gamma}) = \gamma_{id}. \quad (2.24)$$

Finally, we will construct the template to which the samples will be aligned, called elastic mean, $\tilde{\mu}_f = \frac{1}{N}\sum_{i=1}^{N} f_i(0) + \int_0^t \tilde{\mu}_q(s)|\tilde{\mu}_q(s)|ds$, which is the pullback of $\tilde{\mu}_q$ to the original space $\mathcal{F}$. In the Figure 2.21 may be observed the result of the registration of the berkeley growth curves of the Figure 2.6.

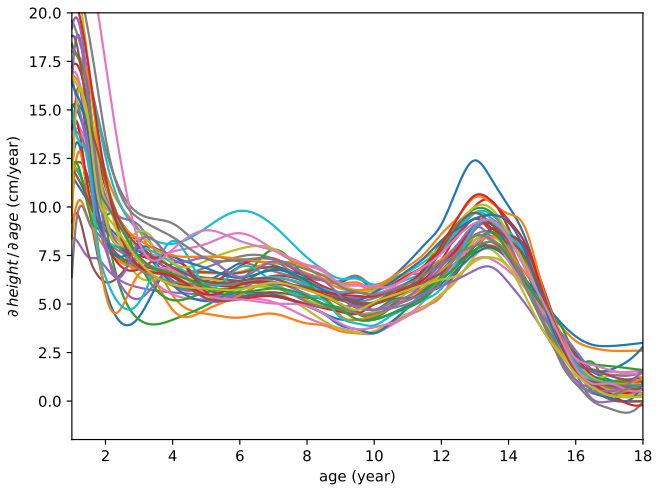


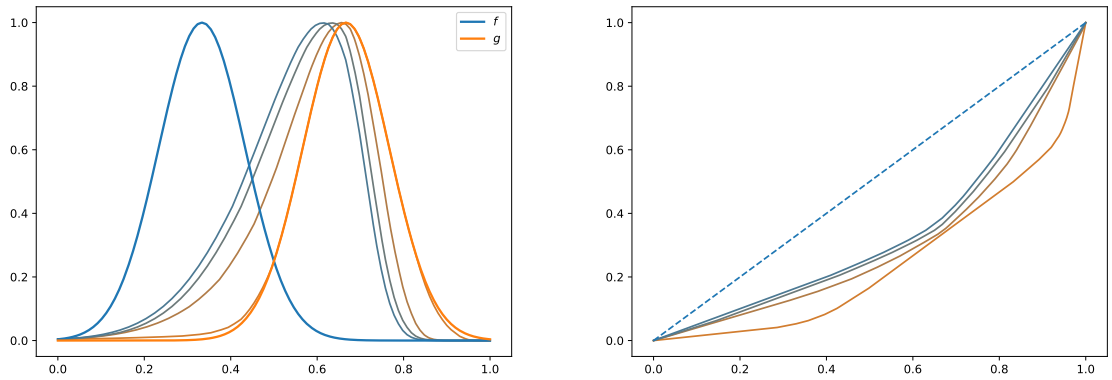
Figure 2.21: Elastic registration of the Berkeley velocity curves

An important property of the elastic mean $\tilde{\mu}_f$ is that given a dataset $\{c_i f(\gamma_i(t)) + e_i\}_{i=1}^n$, where $\{\gamma_i\}_{i=1}^n$ have as Karcher mean in Γ the identity and c_i and e_i are positive constants with mean 1 and 0 respectively, then $\tilde{\mu}_f$ is a consistent estimator of f [15].

2.3.6. Restricting elasticity

Sometimes it is necessary to control the amount of elasticity during the registration process, for this purpose it is possible to add a penalty term $\mathcal{R}(\lambda)$ to the elastic distance 2.16. Let $q_1, q_2 \in \mathbb{L}^2$ be two SRSFs and $\lambda > 0$, we will define the penalized elastic distance as

$$d_\lambda(q_1, q_2) \equiv \inf_{\gamma\in\Gamma} \left(\|q_1 - \left(q_2 \circ \gamma \sqrt{\dot{\gamma}}\right)\|^2 + \lambda \|\sqrt{\dot{\gamma}} - 1\|^2 \right)^{(1/2)}. \quad (2.25)$$



(a) Registration of f_1 to f_2 with a penalty term

(b) Warping used in the alignment

Figure 2.22: Penalized elastic registration with different values of λ

Once the registration process of our data has been completed, we can proceed to another parts in the analysis of the data. Regression and classification are two of the most important problems in statistics, and therefore in FDA. Among the models to address these tasks are the nearest neighbors estimators, which can be generalized directly to functional data. In the following section is made a brief overview of these estimators.

2.4. Nearest neighbors

After representing the data in functional form, and preprocessing it to quantify and reduce its variability, it may be interesting to study the relationship between various functional variables. As in Multivariate Statistics and Machine Learning, FDA will deal with the regression and classification problems, but in the case where some of the data are of a functional nature.

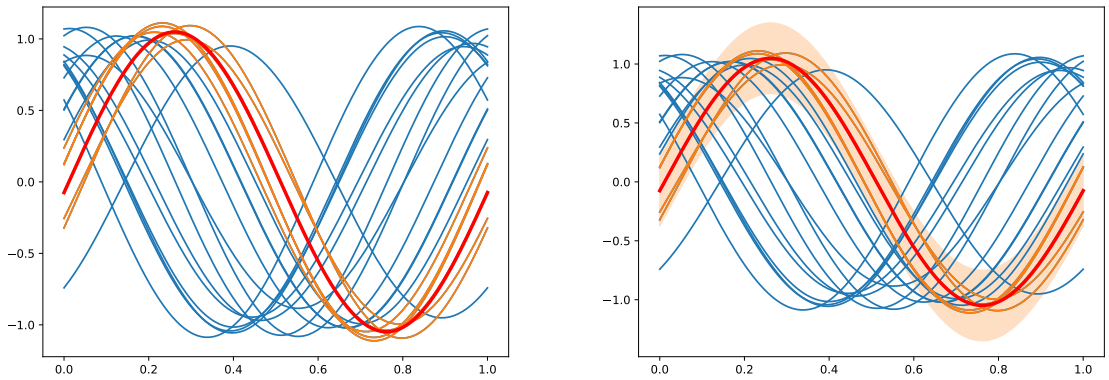
In *classification*, or more precisely supervised classification, the category or population to which an observation belongs is studied, using a set of data whose category is known. For example, predicting the climate type of a region from its temperature curves is a functional classification problem.

In *regression* is studied the existing relation between two or several random variables. Instead of predicting a class label, as in classification, regression models are created to predict another variable, called dependent variable or response, from one or more known variables. In the case of functional prediction we must make a distinction that does not appear in classical statistics between two cases, when the response to predict is an scalar quantity or is also a functional datum. For instance, the prediction of the total precipitation of a region from its temperature curves is a regression problem with scalar response, as opposed to the case with functional response, in which is predicted the precipitation curves from the temperatures. Among the models most used for these two problems can be found the nearest neighbors estimators.

The Nearest Neighbors (NN) estimators are a family of methods widely used in Statistics and Machine Learning, in problems of classification or regression, among others. These estimators are based on the idea of neighborhood, using the notion of distance, so that it is made a local estimation of the density of the data. Although in their classic version they are used with sets of vectors, their ideas work in the same way in general metric spaces [19], as the functional ones we consider in this work.

Let $(\mathcal{F}, d)$ be a metric space and $(f_i, Y_i)_{i \leq i \leq n}$ a training set with their respective labels or responses. To estimate a datum x, either for classification or prediction of its response, firstly, it will be necessary to find the elements of the training set closest to this datum, which will form its neighborhood, denoted as $k(x)$.

There are two variants of these methods. In the first one, consisting of K-Nearest Neighbors (KNN) estimators, it is taken as neighborhood the k closest elements to x, i. e., if the training pairs are re-indexed as $(f_{(i)}, Y_{(i)}) 1 \leq i \leq n$ so that the $f_{(i)}$'s are re-arranged in increasing distance from x, $d(x, f_{(1)}) \leq d(x, f_{(2)}) \leq \cdots \leq d(x, f_{(n)})$, then $k(x) = \{f_{(i)} : 1 \leq i \leq i \leq k\}$.



(a) K-nearest neighbors

(b) Radius-nearest neighbors

Figure 2.23: Neighborhoods using distance $\mathbb{L}^\infty$

In the second variant, less used in practice, consisting of the radius neighbors estimators, the neighborhood contains the samples f_i in the ball of radius r centered in x, i.e. $k(x) = \{f_i : d(f_i, x) \leq r\}$. For instance, if we use the distance $\mathbb{L}^\infty$, we may visualize $k(x)$ as the set of all functions within a band of radius r around x. In the figure 2.23 there are shown the neighborhoods with these two different approaches.

In practice, for the construction of the neighborhoods, the simplest solution is to perform a linear search, calculating the distances between x and all the elements of the training set. The naïve approach of the linear search may be improve using data structures based on spatial indexes, such as ball trees [20]. However, the best nearest-neighbors data structure for a given application will depend on the dimensionality, size, and underlying structure of the data.

2.4.1. Classification

In the classification problem, the samples of the training set f_i are associated to the labels Y_i corresponding to their class. Given a datum x, a new label is predicted using the majority class among its neighbors, so that the predicted class will be

$$\hat{Y} = \underset{j}{\operatorname{argmax}} \sum_{f_i \in k(x)} \mathbb{1}_{\{Y_i = j\}}. \tag{2.26}$$

It is possible to make a weighting vote, so that the closest neighbors will have a greater weight, for example, using $w_i = 1/d(f_i, x)$, so that the resulting label will be $\hat{Y} = \underset{j}{\operatorname{argmax}} \sum_{f_i \in k(x)} w_i \mathbb{1}_{\{Y_i = j\}}$.

2.4.2. Regression

In the regression problem, each of the training samples f_i have a response Y_i associated with them. This response can be either scalar or functional, although the way to proceed will be similar.

In both cases, it is necessary to select the responses associated with the elements of the neighborhood $k(x)$, which will be used to predict the response of the datum x.

In the scalar response case, a weighted average of the neighbors' responses Y_i is used, so that the prediction will be calculated as

$$\hat{Y} = \sum_{(f_i, Y_i): f_i \in k(x)} w_i Y_i \,, \tag{2.27}$$

where $\sum w_i = 1$. which may be chosen based on distance or uniformly.

In the case where the responses are also functional data, the predicted response is constructed in a similar way as in the previous case, using a weighted average of functions, or a centroid, such as the Karcher means presented in Section 2.3.5.

During this chapter we have focused on making a brief description of the mathematical concepts used in this work. The aim of this work was to include all these concepts as functionalities in the *scikit-fda* package. The next chapter summarizes the design decisions followed for this task, the technologies used for it and the methodologies used for this purpose.

3

Design and development

In this chapter we will make a brief summary of the technologies and development followed during this work, in which the mathematical concepts discussed throughout the Section 2 have been incorporated in *scikit-fda*. On the one hand, we will discuss the requirements and design raised in development. During the design stage it has been special emphasis on creating an API similar to the packages of the *scipy* [22] ecosystem, allowing the integration of functionalities of some of these libraries. This facilitates the use of the package to developers and researchers who already know this ecosystem, widely used for Statistics and Machine Learning in Python. Because of these features, and to allow a wider diffusion of the package, it was decided to develop the project as an *scikit*, which are open source packages for scientific computing in Python. To promote this diffusion, during this work has been managed the creation of a logo, used as a symbol of the project. This logo is shown in the Figure 3.1.

Figure 3.1: Scikit-fda logo

On the other hand, this chapter presents the technologies used for the development of the package, which currently supports Python 3.6 and 3.7. Among the tools used in this development has been used *git*, to perform version control. This tool has allowed the communication of the team involved in the development, and the integration of a Continuous integration (CI) system for the execution of tests and generation of online documentation.

In addition, there is a brief summary of the methodology followed in this work, which had to be integrated together with the work of a team. For this integration it has been necessary a great communication, carrying out weekly meetings throughout the year. This communication has been possible thanks to the version control system, which has allowed revisions of the work among the members of

the development team. Due to the open-source nature of the project, these revisions and the code are publicly available. For this reason special emphasis has been placed on the quality of the code, which will be maintained and used by multiple developers throughout the life of the project.

3.1. Analysis

As mentioned above, the *scikit-fda* project started in 2017, at which time an analysis of the requirements of the package was made [5]. These requirements are still in effect today, some of them are:

- The software developed has to be a Python package.
- It has to be an open-source project.
- The software must follow Python standards defined in PEP 8 and PEP 257.
- Documentation has to be intended for a very general audience.
- The project has to include and extensive test bench of unit test and continuous integration mechanism.

In addition to the original ones, three new requirements have been formulated:

- The software should be cross-platform and the mechanism of continuous integration should run the test bench in the main operating systems, that is, Linux, MacOs and Windows, as well as different Python versions supported.
- API should be similar, as far as possible, to the *numpy* [21] , *scipy* and *scikit-learn* ones, allowing whenever possible the use of their functionalities with the objects of the software developed.
- The documentation should contain examples showing different functionalities.

3.2. Design

Originally the package was structured around two classes, *FDataGrid* and *FDataBasis*, designed for each of the main data representations of the data and consisted of four modules with basic statistics, operations and methods to perform kernel smoothing.

Due to the expansion of the project, the package has been completely restructured, with a more hierarchical structure. Figure 3.2 shows a diagram of the main modules of the package. Darker colors in the figure represent a more advanced stage of development of the modules. The following subsections summarize, in general terms, the functionalities incorporated and design changes made during this work. A more detailed description of the functions and classes developed may be found in Annex C, or in the online documentation, available at fda.readthedocs.io.

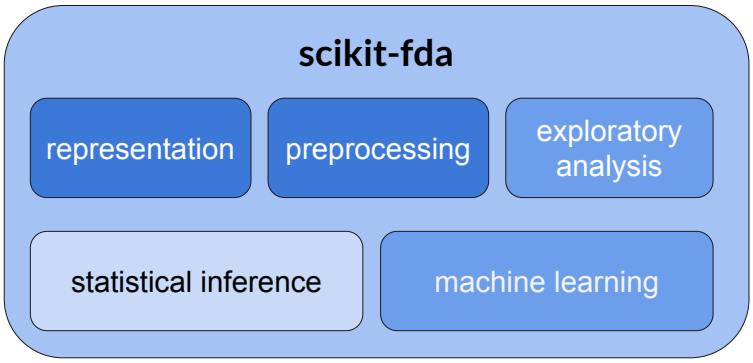


Figure 3.2: Map of scikit-fda [6]

3.2.1. Representation module

The *representation* module includes general functionalities for the representation of functional data and classes for processing this data using a object-oriented paradigm. There were two classes, *FDataBasis* and *FDataGrid*, to represent the data in basis or discretized form, as it was explained in the indroduction of the Chapter 2. These classes contain common functionalities. In order to unify these functionalities, an abstract class called *FData* has been created, from which the previous ones inherit. This class implements methods for the evaluation of the data as functions and its plotting. Aside from methods for the composition of functions or operations between them. The method created for the evaluation of functional data in *FData* allows its evaluation as vectors of functions $f = (f_1, f_2, \ldots, f_n)'$ where $f_i : \mathbb{R}^d \to \mathbb{R}^m$, using a similar syntax to the mathematical notation $f(t)$. It is also allowed to call these functions in a vectorized way with multiple values, so that

$$f((t_1, t_2, \ldots, t_k)) = \begin{pmatrix} f_1((t_1, t_2, \ldots, t_k)) \\ f_2((t_1, t_2, \ldots, t_k)) \\ \vdots \\ f_n((t_1, t_2, \ldots, t_k)) \end{pmatrix} = \begin{pmatrix} f_1(t_1) & f_1(t_2) & \ldots & f_1(t_k) \\ f_2(t_1) & \ddots & & f_2(t_k) \\ \vdots & & \ddots & \vdots \\ f_n(t_1) & f_n(t_2) & \ldots & f_n(t_k) \end{pmatrix} \tag{3.1}$$

where $t_1, t_2, \ldots t_k$ are the points of evaluation in $\mathbb{R}^d$. For example, the code 3.1 shows the creation of a set of three random samples defined in $[0, 1]$. These samples are packed in a *FData* object using a discrete representation. The they are then evaluated at 0, $0{,}5$ and 1.

```
>>> from skfda.datasets import make_sinusoidal_data
>>>
>>> f = make_sinusoidal_process(n_samples=3, start=0, stop=1)
>>> f([0, 0.5, 1]).round(3)
array([[-0.677,  0.368, -0.372],
       [ 0.702, -0.717,  0.765],
       [-0.8  ,  0.405, -0.703]])
```

Code 3.1: Example of evaluation of an *FData*

The evaluation method separates the evaluation points t_j into two sets, depending on whether they are inside or outside the domain range of the functions. The points within the domain range are passed to the evaluator, defined by the type of representation, to its evaluation, which will depend on the implementation of the representation. In the discrete representation it is used interpolation to this purpose.

A submodule called *interpolation* has been created with different interpolation methods, used in the class *FDataGrid* as evaluator. In this module may be found the *SplineInterpolation* class, which implements the different interpolation techniques explained in the Section 2.1.

In addition, *FData* objects have an extrapolator, to which points outside their domain are passed during the evaluation. It was created a submodule called *extrapolation* with different extrapolators used in this task, in which they have included the classes *PeriodicExtrapolation*, to extend the domain periodically, *BoundExtrapolation*, to use the values of the limits or *FillExtrapolation*, to return a fixed value.

Once the data is in functional form using an *FData* object, it is possible to use the rest of the functionalities of the package, such as the registration methods or the models for regression and classification.

3.2.2. Preprocessing module

In the preprocessing module a specific module called *registration* has been created to deal with the registration of the data using the techniques explained throughout the sections 2.2 and 2.3. Among the functions incorporated in this module are *shift_registration*, for registrations using shifts, *landmark_registration*, to employ known landmarks, or *elastic_registration* with the algorithm detailed in Section 2.3.5. In addition, this module contains functions for the treatment of warpings functions among other useful processes during the registration phase. To ensure the efficiency of the algorithm used in the pairwise alignment (eqn. 2.7), it was implemented using *C*, which is called by the Python package through *Cython*.

All the registration methods have been designed to receive an *FData* object and return another one containing the registered samples, which can be used with other functionalities of the package, such as the classification or regression models included in the machine learning module.

3.2.3. Machine learning module

The package contains a module with tools for machine learning. This module is divided into three submodules called *classification*, *regression* and *clustering*. Models for data classification and regression have been incorporated using the nearest neighbors estimators detailed in the Section 2.4.

These models have been designed following the API of the *scikit-learn* estimators, to allow their use

together with the functionalities it offers, such as validation tools or integration in pipelines to automate tasks. Among the classes created in this module are *KNeighborsClassifier*, for classification using KNN, *KNeighborsRegressor*, for both scalar and functional regression, as well as its equivalent using a radius instead of a fixed number of neighbors *RadiusNeighborsClassifier* and *RadiusNeighborsRegressor*. All of these classes contain two main methods, *fit*, to train the model, and *predict*, to predict new labels or responses. In addition, it is possible to change the metrics used for neighbor searches, for this purpose may be used the metrics incorporated in the *metrics* module.

3.2.4. Miscellaneous module

The miscellaneous module module contains different scattered topics used during the analysis of functional data. Specifically, it has been added a *metrics* submodule, which contains functional metrics. These metrics are used in several machine learning models. Among the distances added to this module are the family of $\mathbb{L}^p$ metrics for multivariate functions, the Fisher-Rao distance or elastic metrics based on the framework of the Section 2.3.2, such as the phase or amplitude distance.

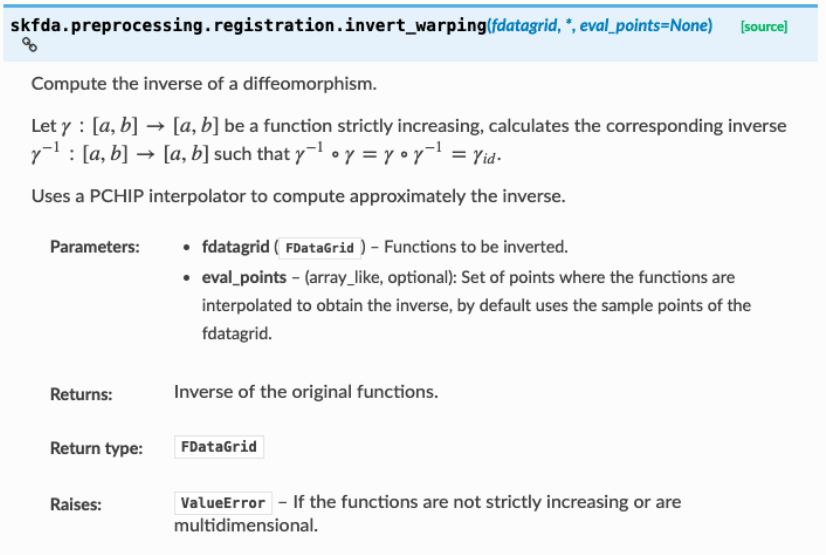
3.2.5. Dataset module

The package contains a module for downloading datasets from different sources as functional objects or the creation of synthetic datasets. Among the methods created for the generation of datasets are included *make_sinusoidal_process*, which generates samples of the form $f_i(t) = a_i \sin(wt+\phi_i)+e_i(t)$, *make_multimodal_samples*, which generates mixtures of gaussian-like functions, such as those of the Figure 2.10(a), or *make_random_warping*, to generate warpings functions like those of the Figure 2.9. These synthetic datasets are useful for the generation of examples and the creation of tests, for this reason they have turned out to be fundamental during the coding and testing phase.

3.3. Coding, documenting and testing

Due to the fact that the project team is composed of several developers, who must review and integrate their work together, it has been necessary to establish a set of common coding standards. Besides, because of the open-source nature of the project, the code is public, and throughout its life will be maintained by multiple developers. For this reason, a great effort had to be dedicated to ensuring the readability of the code, along with its documentation, and compliance with strictly established standards. Among these standards are included two Python Enhancements Proposal (PEP) , which are the directives that set the different conventions in Python: the PEP 8 - *Style Guide for Python Code*, and the PEP 257 - *Docstring Conventions*.

PEP 257 documents the semantics and conventions associated with Python docstrings, which allow include documentation within the code. These docstring are found next to the code as headers of modules, classes or functions. It has been used a google-like docstring style, based in the *reStructuredText* format. This style allows the automatic generation of documentation in pdf and html, using the *Sphinx* tool. Figure 3.3 shows an extract of the resulting documentation. These pages are generated automatically by the *readthedocs* platform and uploaded to fda.readthedocs.io.



skfda.preprocessing.registration.invert_warping(*fdatagrid*, **, eval_points=None*) [source]

Compute the inverse of a diffeomorphism.

Let $\gamma : [a, b] \to [a, b]$ be a function strictly increasing, calculates the corresponding inverse $\gamma^{-1} : [a, b] \to [a, b]$ such that $\gamma^{-1} \circ \gamma = \gamma \circ \gamma^{-1} = \gamma_{id}$.

Uses a PCHIP interpolator to compute approximately the inverse.

Parameters:
- **fdatagrid** (`FDataGrid`) – Functions to be inverted.
- **eval_points** – (array_like, optional): Set of points where the functions are interpolated to obtain the inverse, by default uses the sample points of the fdatagrid.

Returns: Inverse of the original functions.

Return type: `FDataGrid`

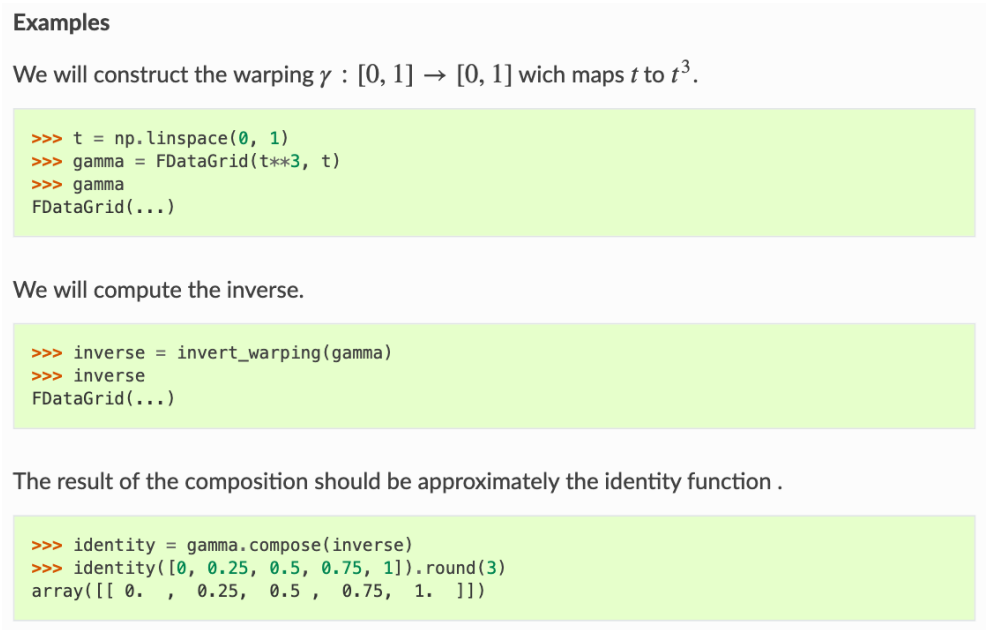
Raises: `ValueError` – If the functions are not strictly increasing or are multidimensional.

Figure 3.3: Scikit-fda online documentation

In addition, among the advantages of this kind of documentation is the possibility of including examples embedded in it, called *doctests*, which appears as dynamic short examples within the documentation, as it is shown in the Figure 3.4. These examples are in turn tests. When running the bench tests, using the tool pytest, the code is parsed looking for *doctests*, executing the code found in them and checking that the output matches with the output of the documentation.

However, the fundamental part of the testing is made up of unit tests, which are executed together with the *doctests* to check the integrity of the software. These unit tests check each of the functions developed separately. In order to measure the coverage of these tests, the tool *Coverage* was used, which allows quantifying the lines executed during these tests.

A very relevant part of the package documentation is made up of examples, which are Python notebooks written as tutorials. Due to their extension, these examples can be found in Annex B or among the online documentation at fda.readthedocs.io/en/latest/auto_examples/. These examples show simple use cases of the package, linking with the documentation of the functions used. Among the examples created in this work can be found some showing basic functionalities of the representation, such as the composition of functions, interpolation or extrapolation. Examples have also been created by showing the recording techniques mentioned in the Section 2.2, or by using the estimators of nearest neighbours in several simple examples.



Examples

We will construct the warping $\gamma : [0, 1] \to [0, 1]$ wich maps t to t^3.

```
>>> t = np.linspace(0, 1)
>>> gamma = FDataGrid(t**3, t)
>>> gamma
FDataGrid(...)
```

We will compute the inverse.

```
>>> inverse = invert_warping(gamma)
>>> inverse
FDataGrid(...)
```

The result of the composition should be approximately the identity function .

```
>>> identity = gamma.compose(inverse)
>>> identity([0, 0.25, 0.5, 0.75, 1]).round(3)
array([[ 0.  ,  0.25,  0.5 ,  0.75,  1.  ]])
```

Figure 3.4: Doctest

3.4. Development, version control and continuous integration

During the development of the package has been used an agile methodology, with similar precepts than the eXtreme programming [23], which is based on simplicity in development, continuous feedback and communication between team members. This communication has been possible by means of weekly meetings held throughout the year.

For this communication in conjunction with version control *git* has been using, along with the functionalities available in *Github*. Currently, the package is hosted in the repository https://github.com/GAA-UAM/scikit-fda. This platform offers a series of tools for the review and control of contributions. There are multiple workflows designed to carry out version control with *git*, of which we employ *Gitflow*.

Gitflow is a widely used workflow, which uses *git* branches to organize the work. It structures the code around two main branches: master and develop. The first is a stable branch, where any built-in commit must be ready to go into production and generates a new version. The second one, develop, is the code that will make up the next planned version of the project. To add new code to these branches, it is necessary to create other auxiliary ones that will be merged to develop by means of a pull request, after a review process and pass the bench tests. Figure 3.5 illustrates the mechanics of this workflow.

One of the main advantages of using *Github*, over other version control systems, is the possibility of using a CI mechanism. *Travis CI* has been used for this purpose, which compiles the library and runs the bench tests each time a commit is made. This has made the development task much easier due to

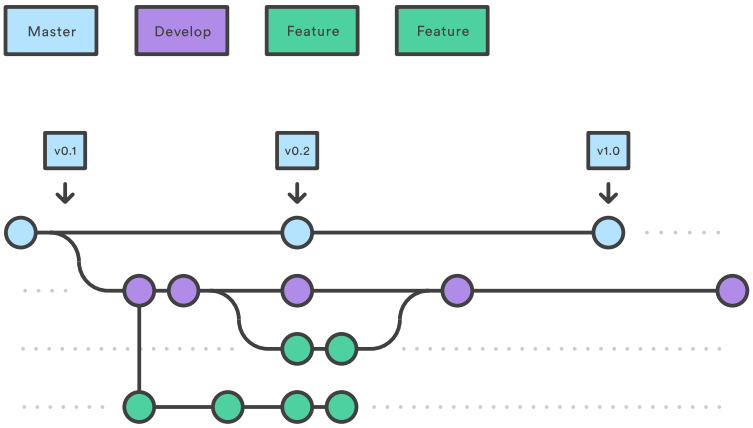


Figure 3.5: Example of git flow branches [1]

the size of the project, reducing the occurrence of bugs in the code. In addition, *Travis CI* has made it possible to automate a wide variety of tasks, such as compiling or integrating style reviewers.

[1] Picture from https://es.atlassian.com/git, licensed under a Creative Commons Attribution 2.5 Australia License (accesed on June 2019)

4

Conclusions and future work

During this year, the project has evolved drastically, redesigning and expanding its functionalities. As a result of this work, the *scikit-fda* package incorporates interpolation techniques, which allow the evaluation of functional data as continuous functions. In addition, a wide variety of data registration techniques and elastic methods useful in FDA have been included. Through work with nearest neighbors estimators, the package provides models to address the classification and regression problems with functional data.

Among the aspects that have allowed this development to be successfully achieved are communication between team members, with weekly meetings, and the use of an CI system which has made it much easier to integrate working together. In addition, the experience of the team throughout the last year in which the project was initiated has been fundamental.

In spite of all the advances made during this year there is still a lot of work to be done. It would be interesting to expand the basis representations of functional data, including support for surfaces, so that covariances can be represented in this form, and provide support to more types of basis, such as wavelets [24] or constrained basis [7]. It would also be interesting to extend the registration techniques, incorporating functionalities from the *fdasrvf* [17] package, which include tools for clustering and registration, or for elastic alignment of spatial curves and surfaces.

Leaving aside these technical aspects, one of the most important tasks to carry out is to give visibility to the project, discovering the library to interested developers and researchers, presenting the package in congresses and publishing articles to promote its use.

To conclude, I would like to make a brief summary of the skills acquired during the course of this work. On the one hand, this work has allowed me to learn concepts of functional data analysis, as well as knowledge of scientific programming and the requirements and design that software of this type must meet. On the other hand, due to the collaboration together with a team, I have been able to learn dynamics of work for the development of software, and for the collaboration in team projects. All this makes this work personally worthwhile, and has served to acquire a global perspective of the knowledge acquired in this degree.

Bibliography

[1] J. O. Ramsay, "When the data are functions," *Psychometrika*, vol. 47, no. 4, pp. 379–396, 1982.

[2] J. O. Ramsay, H. Wickham, S. Graves, and G. Hooker, "Cran R - fda package."

[3] J. O. Ramsay, G. Hooker, and S. Graves, *Functional Data Analysis with R and MATLAB*. Springer, 1 ed., 2009.

[4] M. Febrero-Bande, "Statistical Computing in Functional Data Analysis: The R Package fda.usc," *Journal of Statistical Software*, vol. 51, no. 4, pp. 1–28, 2012.

[5] M. Carbajo, "FDA-PY: Development of a python package for functional data analysis," 2018.

[6] C. Ramos-Carreño, "Scikit-fda: A Python package for Functional Data Analysis." III International Workshop on Advances in Functional Data Analysis, 05 2019.

[7] J. O. Ramsay and B. W. Silverman, *Functional Data Analysis*. 2 ed., 2005.

[8] A. Srivastava and E. Klassen, *Functional and Shape Data Analysis*, vol. 49. Springer, 2016.

[9] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. Vanderplas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," pp. 1–15, 2013.

[10] C. R. de Boor, "B(asic)-Spline Basics," vol. 3, no. September, 1981.

[11] H. E. Jone and N. Bayley, "The Berkeley Growth Study," *Child Development*, vol. 12, no. Jun, pp. 167–173, 1941.

[12] D. Bertsekas, "Dynamic Programming and Optimal Control," 1995.

[13] J. S. Marron, J. O. Ramsay, L. M. Sangalli, and A. Srivastava, "Functional Data Analysis of Amplitude and Phase Variation," *Statistical Science*, vol. 30, no. 4, pp. 468–484, 2015.

[14] A. Kneip and J. O. Ramsay, "Combining registration and fitting for functional models," *Journal of the American Statistical Association*, vol. 103, no. 483, pp. 1155–1165, 2008.

[15] A. Srivastava, W. Wu, S. Kurtek, E. Klassen, and J. S. Marron, "Registration of Functional Data Using Fisher-Rao Metric," no. March, 2011.

[16] J. D. Tucker, *Functional Component Analysis and Regression Using Elastic Methods*. PhD thesis, Florida State University, 2014.

[17] J. D. Tucker, "fdasrvf: Elastic Functional Data Analysis," 2017.

[18] N. Cêncov, "Statistical Decision Rules and Optimal Inferences," *Translations of Mathematical Monographs*, vol. 53, 1982.

[19] A. Baíllo, A. Cuevas, and R. Fraiman, "Classification methods for functional data," in *The Oxford Handbook of Functional Data Analysis* (F. Ferraty and Y. Romain, eds.), ch. 10, pp. 259–297, Oxford University Press, 2010.

[20] N. Kumar, L. Zhang, and S. Nayar, "What is a good nearest neighbors algorithm for finding similar patches in images?," *Lecture Notes in Computer Science (including subseries Lecture Notes in*

Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 5303 LNCS, no. PART 2, pp. 364–378, 2008.

[21] T. Oliphant, “NumPy: A guide to NumPy.” USA: Trelgol Publishing, 2006–.

[22] E. Jones, T. Oliphant, and P. Peterson, “SciPy: Open source scientific tools for Python.”

[23] J. Blankenship, M. Bussa, and S. Millet, “eXtreme Programming,” in *Pro Agile .NET Development with Scrum*, ch. 3, 2011.

[24] P. A. Morettin, A. Pinheiro, and B. Vidakovic, *Wavelets in Functional Data Analysis*. Springer, 2017.

Acronyms

API Application Programming Interface.

CI Continuous integration.

DPA Dinamic programming algorithm.

FDA Functional data analysis.

KNN K-Nearest Neighbors.

NN Nearest Neighbors.

PEP Python Enhancements Proposal.

REGSSE Registered sum of squared errors.

SRSF Square Root Slope Funcion.

Appendices

A

Algorithms and proofs

A.1. Shift registration by the Newton-Raphson algorithm

For the calculation of the values δ_i necessary for the shift registration of a set $\{f_i\}_{i=1}^n$, according to 2.3, we will use a variation of the Newton-Rhapson's root-finding algorithm, applied to the derivative of REGSSE (2.4). This procedure is explained in more detail in Ramsay and Silverman (2005) [7].

For this computation it will be necessary to evaluate derivatives of f_i, so it will be crucial a previous smoothing step of the samples. Derivatives of REGGSE are given by:

$$\frac{\partial}{\partial \delta_i}\text{REGSSE} = 2\int_{\mathcal{T}} \left[f_i(t+\delta_i) - \hat{\mu}(t)\right] Df_i(t)dt, \tag{A.1}$$

$$\frac{\partial^2}{\partial \delta_i^2}\text{REGSSE} = 2\int_{\mathcal{T}} \left[f_i(t+\delta_i) - \hat{\mu}(t)\right] D^2 f_i(t)dt + 2\int_{\mathcal{T}} \left[Df_i(t)^2\right] dt. \tag{A.2}$$

In practice the first term of $\frac{\partial^2}{\partial \delta_i^2}$**REGSSE** (A.2) it is deleted, because when the misalignment of the samples is large it can affect the convergence of the algorithm, and vanishes for the values that minimize the criterion. Therefore the following approximation is used:

$$\frac{\partial^2}{\partial \delta_i^2}\text{REGSSE} \approx 2\int_{\mathcal{T}} \left[Df_i(t)^2\right] dt. \tag{A.3}$$

The initilization of $\delta_i^{(0)}$ in A.1 may be set to minimize some feature, or simply set $\delta_i^{(0)}$ to 0.

Could be used as stop criterion a maximun value of iterations along with a tolerance $|\delta_i^{(\nu)} - \delta_i^{(\nu-1)}| < \epsilon$. Generally the convergence is fast, obtaining good alignments with one or two iteration with a reasonable estimation of the initial values $\delta_i^{(0)}$. The step size α in A.1 may be set to 1.

Input : Set of functional observations $\{f_i(t)\}_{i=1}^n$
Output: Shifts $\{\delta_i\}_{i=1}^n$ used to register the data

Initialize $\delta^{(0)}$;
for *step* $\nu = 1, 2, \ldots$ *until stop criterion* **do**
  Update cross-sectional mean
  $\hat{\mu}(t) \leftarrow \frac{1}{n}\sum_{i=1}^n f_i(t + \delta_i^{(\nu-1)})$
  foreach $\delta_i^{(\nu-1)}$ **do**
  Update values of δ_i
  $\delta_i^{(\nu)} \leftarrow \delta_i^{(\nu-1)} - \alpha \frac{\partial}{\partial \delta_i} REGSSE / \frac{\partial^2}{\partial \delta_i^2} REGSSE$;
end

Algorithm A.1: Shift registration by Rhapson-Newton algorithm

A.2. Proofs of some mathematical results

In this section two fundamental results are proved, referenced in the section 2.3. These proofs have been extracted from [8] and [15]

Lemma 1: *Under the SRVF representation, the Fisher-Rao Riemannian metric becomes the standard metric.*

We can decompose the SRSF mapping into two steps: $f(t) \to \dot{f}(t) \to q(t) = sign(\dot{f})\sqrt{\dot{f}} = Q(\dot{f})$.

For any $v \in T_f(\mathcal{F})$, the differential of this mapping is $v(t) \to \dot{v}(t) \to w(t) = Q_{*,f(t)}(\dot{v}(t))$. To evaluate the expression for w, we need the expression for Q_*. In case $x > 0$, we have $Q(x) = \sqrt{x}$ and its directional derivative in the direction of $y \in \mathbb{R}$ is $y/(2\sqrt{x})$. In case $x < 0$, we have $Q(x) = -\sqrt{-x}$ and its directional derivative is $y/(2\sqrt{-x})$. Combining the two, the directional derivative of Q is $Q_{*,x(y)} = y/(2\sqrt{|x|})$.

Let $v_1, v_2 \in T_f(\mathcal{F})$ be two vectors in the tangent space, and their mappings as $w_i(t) = \dot{v}_i(t)/(2\sqrt{|\dot{f}(t)|})$. Taking the $\mathbb{L}^2$ inner-product between the resulting tangent vectors, we get:

$$\langle w_1(t), w_2(t)\rangle = \int_0^1 w_1(t)w_2(t)dt = \frac{1}{4}\int_0^1 \dot{v}_1(t)\dot{v}_2(t)\frac{1}{|f(t)|}dt$$

The RHS is compared with Eqn.2.12 to complete the proof.

Lemma 2: *For any two SRVFs* $q1, q2 \in \mathbb{L}^2$ *and* $\gamma \in \Gamma$*, we have that* $\|(q_1, \gamma) - (q_2, \gamma)\| = \|q_1 - q_2\|$.

Let $\gamma \in \Gamma$ be and arbitrary warping and $q1, q2 \in \mathbb{L}^2$,

$$\|(q_1, \gamma) - (q_2, \gamma)\|^2 = \int_0^1 \left(q_1(\gamma(t))\sqrt{\dot{\gamma}(t)} - q_2(\gamma(t))\sqrt{\dot{\gamma}(t)}\right)^2 dt =$$

$$\int_0^1 \left(q_1(\gamma(t)) - q_2(\gamma(t))\right)^2 \dot{\gamma}(t)dt = \|q_1 - q_2\|^2 .\square$$

B

Example notebooks

This appendix includes examples showing the use of different functionalities implemented in the package. These *Python notebooks* are available together with the online documentation.

The examples included in this annex are:

1.– Interpolation

2.– Extrapolation

3.– Composition

4.– Shift registration

5.– Landmark shift

6.– Landmark registration

7.– Pairwise alignment

8.– Elastic registration

9.– K-nearest neighbors classification

10.– Radius-nearest neighbors classification

11.– Neighbors scalar regressions

Note

Click here to download the full example code

Interpolation

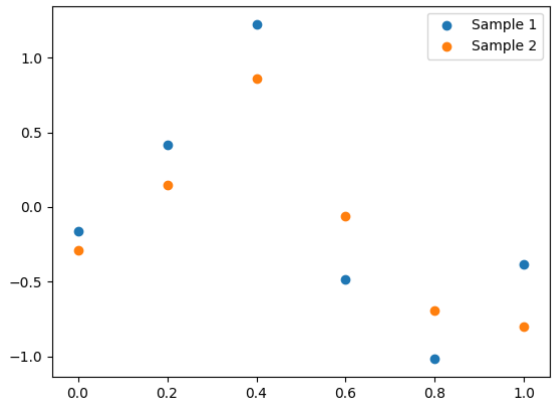
This example shows the types of interpolation used in the evaluation of FDataGrids.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

import skfda
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from skfda.representation.interpolation import SplineInterpolator
```

The `FDataGrid` class is used for datasets containing discretized functions. For the evaluation between the points of discretization, or sample points, is necessary to interpolate.

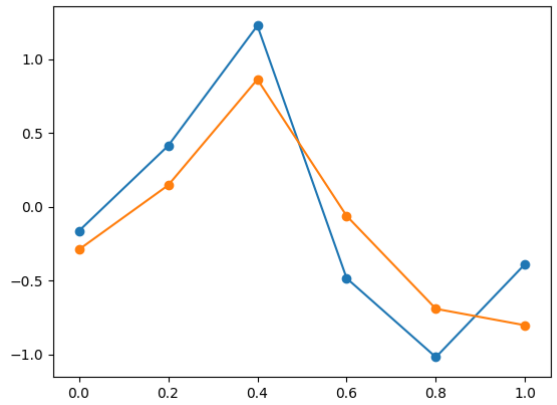
We will construct an example dataset with two curves with 6 points of discretization.

```
fd = skfda.datasets.make_sinusoidal_process(n_samples=2, n_features=6,
                                            random_state=1)
fd.scatter()
plt.legend(["Sample 1", "Sample 2"])
```



By default it is used linear interpolation, which is one of the simplest methods of interpolation and therefore one of the least computationally expensive, but has the disadvantage that the interpolant is not differentiable at the points of discretization.

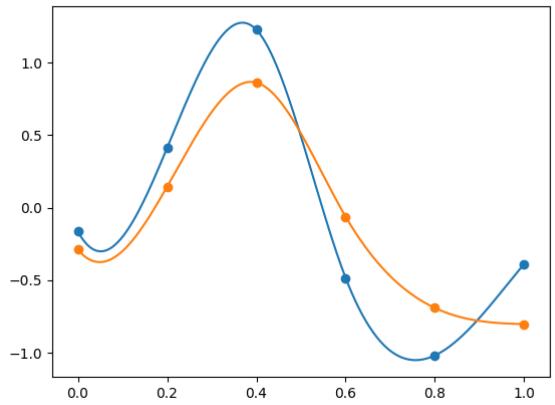
```
fd.plot()
fd.scatter()
```



The interpolation method of the FDataGrid could be changed setting the attribute *interpolator*. Once we have set an interpolator it is used for the evaluation of the object.

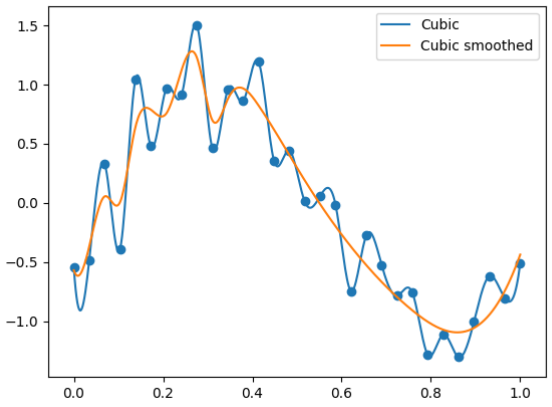
Polynomial spline interpolation could be performed using the interpolator `SplineInterpolator`. In the following example a cubic interpolator is set.

```
fd.interpolator = SplineInterpolator(interpolation_order=3)

fd.plot()
fd.scatter()
```



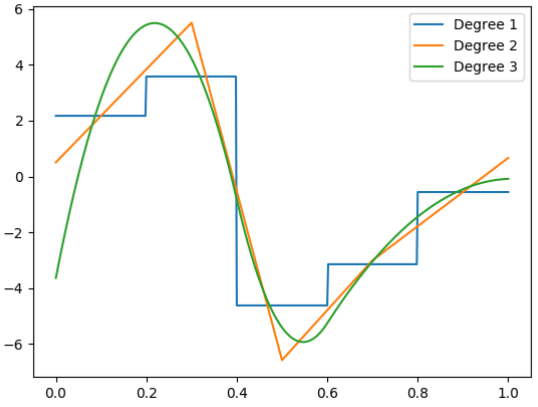
Smooth interpolation could be performed with the attribute *smoothness_parameter* of the spline interpolator.

```
# Sample with noise
fd_smooth = skfda.datasets.make_sinusoidal_process(n_samples=1, n_features=30,
                                                   random_state=1, error_std=.3)

# Cubic interpolator
fd_smooth.interpolator = SplineInterpolator(interpolation_order=3)

fd_smooth.plot(label="Cubic")

# Smooth interpolation
fd_smooth.interpolator = SplineInterpolator(interpolation_order=3,
                                            smoothness_parameter=1.5)

fd_smooth.plot(label="Cubic smoothed")

fd_smooth.scatter()
plt.legend()
```



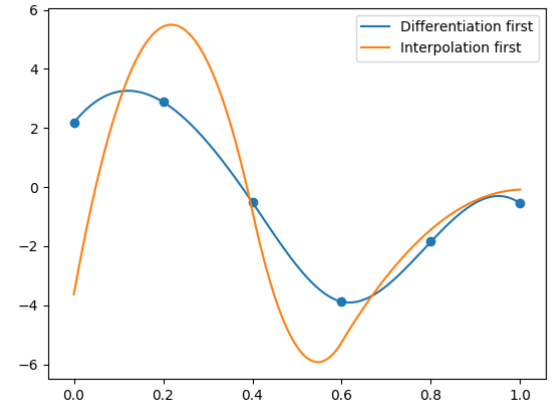
It is possible to evaluate derivatives of the FDatagrid, but due to the fact that interpolation is performed first, the interpolation loses one degree for each order of derivation. In the next example, it is shown the first derivative of a sample using interpolation with different degrees.

```
fd = fd[1]

for i in range(1, 4):
    fd.interpolator = SplineInterpolator(interpolation_order=i)
    fd.plot(derivative=1, label=f"Degree {i}")

plt.legend()
```



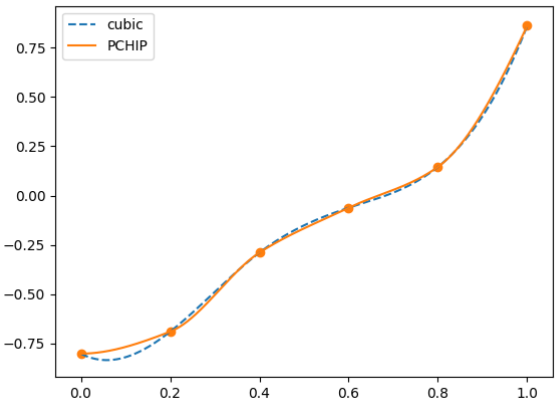
FDataGrids can be differentiate using lagged differences with the method `derivative()`, creating another FDataGrid which could be interpolated in order to avoid interpolating before differentiating.

```
fd_derivative = fd.derivative()

fd_derivative.plot(label="Differentiation first")
fd_derivative.scatter()

fd.plot(derivative=1, label="Interpolation first")

plt.legend()
```



Sometimes our samples are required to be monotone, in these cases it is possible to use monotone cubic interpolation with the attribute *monotone*. A piecewise cubic hermite interpolating polynomial (PCHIP) will be used.

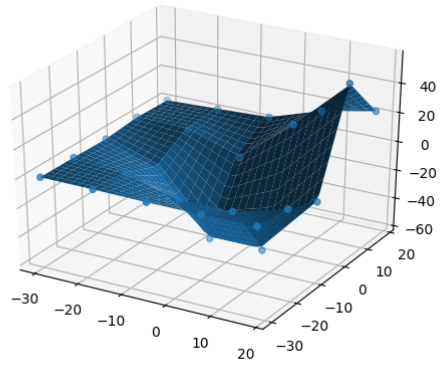
```
fd_monotone = fd.copy(data_matrix=np.sort(fd.data_matrix, axis=1))


fd_monotone.plot(linestyle='--', label="cubic")



fd_monotone.interpolator = SplineInterpolator(interpolation_order=3,
                                              monotone=True)
fd_monotone.plot(label="PCHIP")

fd_monotone.scatter(c='C1')
plt.legend()
```



All the interpolators will work regardless of the dimension of the image, but depending on the domain dimension some methods will not be available.

For the next examples it is constructed a surface, $x_i : \mathbb{R}^2 \longmapsto \mathbb{R}$. By default, as in unidimensional samples, it is used linear interpolation.

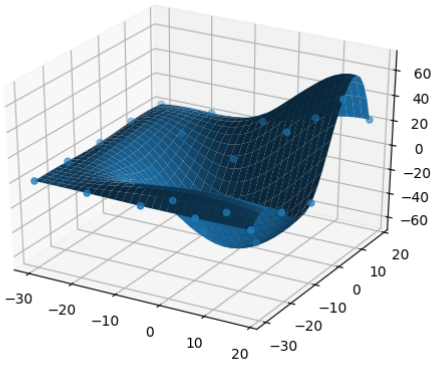
```
X, Y, Z = axes3d.get_test_data(1.2)
data_matrix = [Z.T]
sample_points = [X[0,:], Y[:, 0]]


fd = skfda.FDataGrid(data_matrix, sample_points)

fig, ax = fd.plot()
fd.scatter(ax=ax)
```



In the following figure it is shown the result of the cubic interpolation applied to the surface.

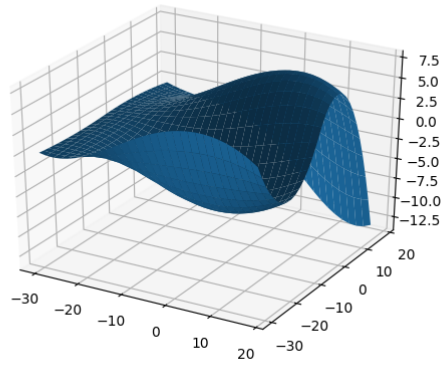
The degree of the interpolator polynomial does not have to coincide in both directions, for example, cubic interpolation in the first component and quadratic in the second one could be defined using a tuple with the values (3,2).

```
fd.interpolator = SplineInterpolator(interpolation_order=3)

fig, ax = fd.plot()
fd.scatter(ax=ax)

plt.show()
```



In case of surface derivatives could be taked in two directions, for this reason a tuple with the order of derivates in each direction could be passed. Let $x(t,s)$ be the surface, in the following example it is shown the derivative with respect to the second coordinate, $\frac{\partial}{\partial s}x(t,s)$.

```
fd.plot(derivative=(0, 1))

plt.show()
```



The following table shows the interpolation methods available by the class `SplineInterpolator` depending on the domain dimension.

Domain dimension	Linear	Up to degree 5	Monotone	Derivatives	Smoothing
1	✔	✔	✔	✔	✔
2	✔	✔	✖	✔	✔
3 or more	✔	✖	✖	✖	✖

Total running time of the script: (0 minutes 2.082 seconds)

Download Python source code: plot_interpolation.py

Download Jupyter notebook: plot_interpolation.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

Extrapolation

Shows the usage of the different types of extrapolation.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 2

import skfda
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d
```

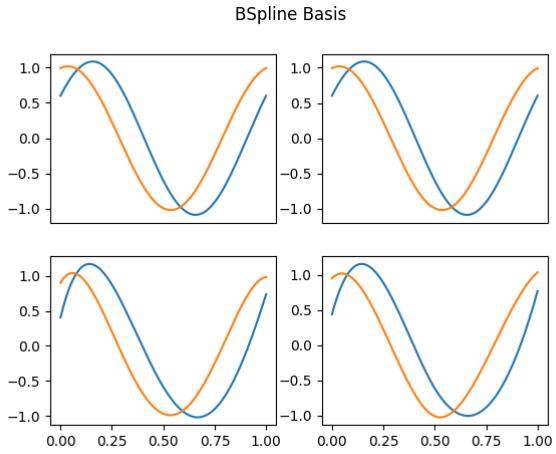
The extrapolation defines how to evaluate points that are outside the domain range of a `FDataBasis` or a `FDataGrid`.

The `FDataBasis` objects have a predefined extrapolation which is applied in ´evaluate´ if the argument *extrapolation* is not supplied. This default value could be specified when the object is created or changing the attribute *extrapolation*.

The extrapolation could be specified by a string with the short name of an extrapolator, with an :class:´Extrapolator <skfda.Extrapolator>´ or with a callable.

To show how it works we will create a dataset with two unidimensional curves defined in (0,1), and we will represent it using a grid and different types of basis.

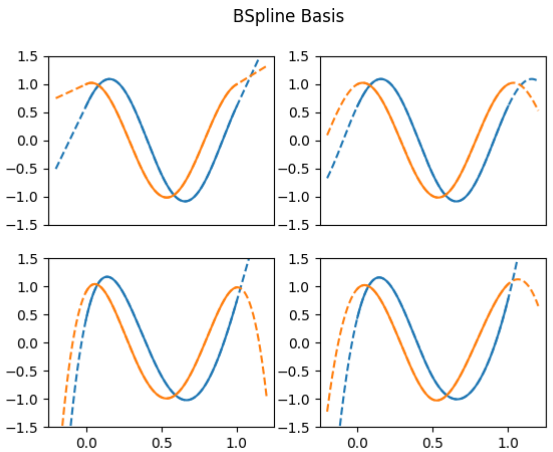
```
fdgrid = skfda.datasets.make_sinusoidal_process(n_samples=2, error_std=0,
random_state=0)
fdgrid.dataset_label = "Grid"

fd_fourier = fdgrid.to_basis(skfda.representation.basis.Fourier())
fd_fourier.dataset_label = "Fourier Basis"

fd_monomial = fdgrid.to_basis(skfda.representation.basis.Monomial(nbasis=5))
fd_monomial.dataset_label = "Monomial Basis"

fd_bspline = fdgrid.to_basis(skfda.representation.basis.BSpline(nbasis=5))
fd_bspline.dataset_label = "BSpline Basis"


# Plot of diferent representations
fig, ax = plt.subplots(2,2)
fdgrid.plot(ax[0][0])
fd_fourier.plot(ax[0][1])
fd_monomial.plot(ax[1][0])
fd_bspline.plot(ax[1][1])

# Disable xticks of first row
ax[0][0].set_xticks([])
ax[0][1].set_xticks([])

# Clear title for next plots
fdgrid.dataset_label = ""
```



If the extrapolation is not specified when a list of points is evaluated and the default extrapolation of the objects has not been specified it is used the type *"none"*, which will evaluate the points outside the domain without any kind of control.

For this reason the behavior outside the domain will change depending on the representation, obtaining a periodic behavior in the case of the Fourier basis and polynomial behaviors in the rest of the cases.

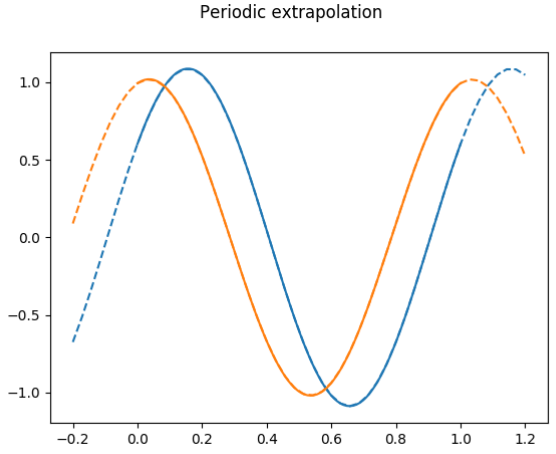
```
domain_extended = (-0.2, 1.2)

fig, ax = plt.subplots(2,2)


# Plot objects in the domain range extended
fdgrid.plot(ax[0][0], domain_range=domain_extended, linestyle='--')
fd_fourier.plot(ax[0][1],domain_range=domain_extended, linestyle='--')
fd_monomial.plot(ax[1][0], domain_range=domain_extended, linestyle='--')
fd_bspline.plot(ax[1][1], domain_range=domain_extended, linestyle='--')

# Plot configuration
for axes in fig.axes:
    axes.set_prop_cycle(None)
    axes.set_ylim((-1.5,1.5))
    axes.set_xlim((-0.25,1.25))

# Disable xticks of first row
ax[0][0].set_xticks([])
ax[0][1].set_xticks([])

# Plot objects in the domain range
fdgrid.plot(ax[0][0])
fd_fourier.plot(ax[0][1])
fd_monomial.plot(ax[1][0])
fd_bspline.plot(ax[1][1])
```



Periodic extrapolation will extend the domain range periodically. The following example shows the periodical extension of an FDataGrid.

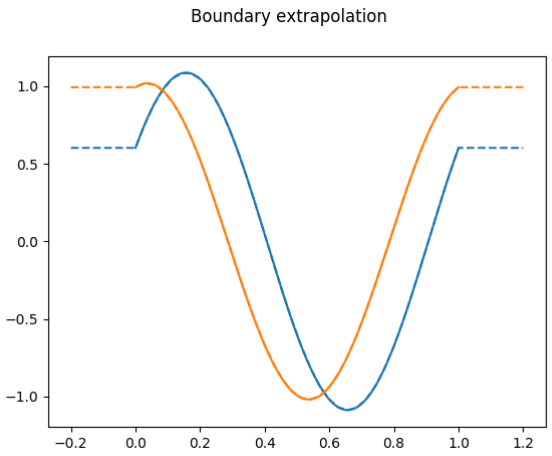
It should be noted that the Fourier basis is periodic in itself, but the period does not have to coincide with the domain range, obtaining different results applying or not extrapolation in case of not coinciding.

```
t = np.linspace(*domain_extended)

plt.figure()
fdgrid.dataset_label = "Periodic extrapolation"

# Evaluation of the grid
# Extrapolation supplied in the evaluation
values = fdgrid(t, extrapolation="periodic")

plt.plot(t, values.T, linestyle='--')

plt.gca().set_prop_cycle(None) # Reset color cycle

fdgrid.plot() # Plot dataset
```



Another possible extrapolation, "bounds", will use the values of the interval bounds for points outside the domain range.

```
plt.figure()
fdgrid.dataset_label = "Boundary extrapolation"

# Other way to call the extrapolation, changing the default value
fdgrid.extrapolation = "bounds"

# Evaluation of the grid
values = fdgrid(t)
plt.plot(t, values.T, linestyle='--')

plt.gca().set_prop_cycle(None) # Reset color cycle

fdgrid.plot() # Plot dataset
```



The :class:´FillExtrapolation <skfda.FillExtrapolation>´ will fill the points extrapolated with the same value. The case of filling with zeros could be specified with the string ` “zeros”` , which is equivalent to *extrapolation=FillExtrapolation(0)*.

```
try:
    res = fd_fourier(t, extrapolation="exception")

except ValueError as e:
    print(e)
```

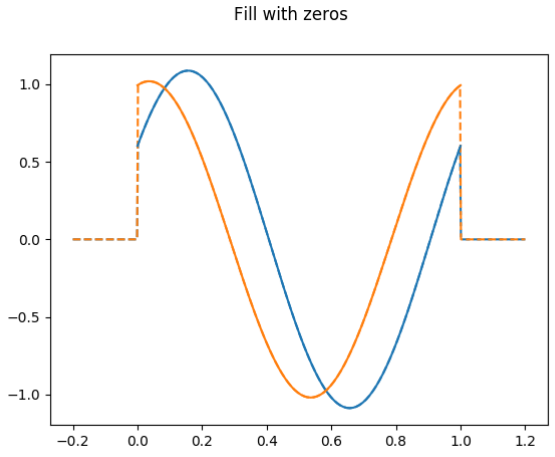
Out:

```
Attempt to evaluate 15 points outside the domain range.
```

All the extrapolators shown will work with multidimensional objects. In the following example it is constructed a 2d-surface and it is extended using periodic extrapolation.

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Make data.
t = np.arange(-2.5, 2.75, 0.25)
X, Y = np.meshgrid(t, t)
Z = np.exp(-0.5 * (X**2 + Y**2))

# Creation of FDataGrid
fd_surface = skfda.FDataGrid([Z], (t, t))

t = np.arange(-7, 7.5, 0.5)

# Evaluation with periodic extrapolation
values =  fd_surface((t,t), grid=True, extrapolation="periodic")
T, S = np.meshgrid(t, t)


ax.plot_wireframe(T, S, values[0], alpha=.3, color="C0")
ax.plot_surface(X, Y, Z, color="C0")
```

```
plt.figure()
fdgrid.dataset_label = "Fill with zeros"

# Evaluation of the grid filling with zeros
fdgrid.extrapolation = "zeros"

# Plot in domain extended
fdgrid.plot(domain_range=domain_extended, linestyle='--')

plt.gca().set_prop_cycle(None) # Reset color cycle

fdgrid.plot() # Plot dataset
```



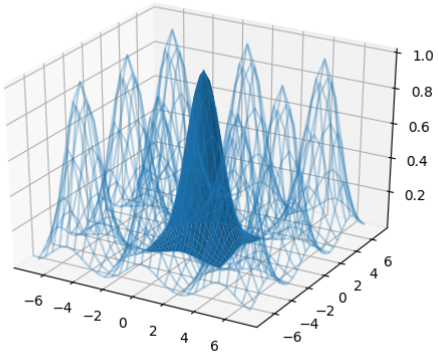
The string “nan” is equivalent to *FillExtrapolation(np.nan)*.

```
values = fdgrid([-1, 0, 0.5, 1, 2], extrapolation="nan")
print(values)
```

Out:

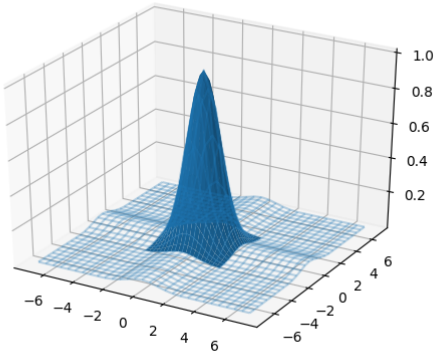
```
[[        nan  0.60293807 -0.60263451  0.60293807         nan]
 [        nan  0.99401003 -0.99350959  0.99401003         nan]]
```

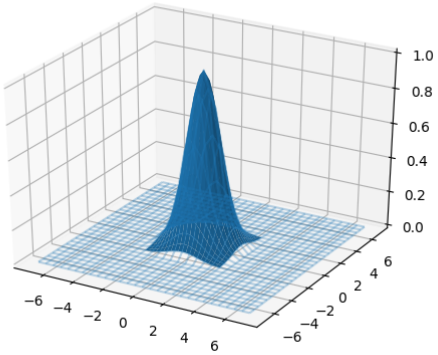
It is possible to configure the extrapolation to raise an exception in case of evaluating a point outside the domain.



The previous extension can be compared with the extrapolation using the values of the bounds.

```
values =  fd_surface((t,t), grid=True, extrapolation="bounds")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(T, S, values[0], alpha=.3, color="C0")
ax.plot_surface(X, Y, Z, color="C0")
```





Or filling the surface with zeros outside the domain.

```
values =  fd_surface((t,t), grid=True, extrapolation="zeros")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_wireframe(T, S, values[0], alpha=.3, color="C0")
ax.plot_surface(X, Y, Z, color="C0")
```

Total running time of the script: (0 minutes 1.807 seconds)

Download Python source code: plot_extrapolation.py

Download Jupyter notebook: plot_extrapolation.ipynb

Gallery generated by Sphinx-Gallery

ï“ Note

Click here to download the full example code

Function composition

This example shows the composition of multidimensional FDataGrids.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

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

from mpl_toolkits.mplot3d import axes3d
```

Function composition can be applied to our data once is in functional form using the method `compose()`.

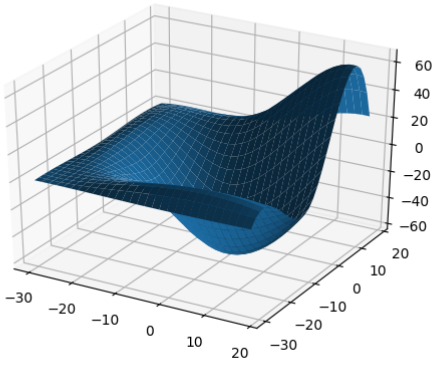
Let $f : X \to Y$ and $g : Y \to Z$, the composition will produce a third function $g \circ f : X \to Z$ which maps $x \in X$ to $g(f(x))$ [1].

In Landmark Registration it is shown the simplest case, where it is used to apply a transformation of the time scale of unidimensional data to register its features.

The following example shows the basic usage applied to a surface and a curve, although the method will work for data with arbitrary dimensions to.

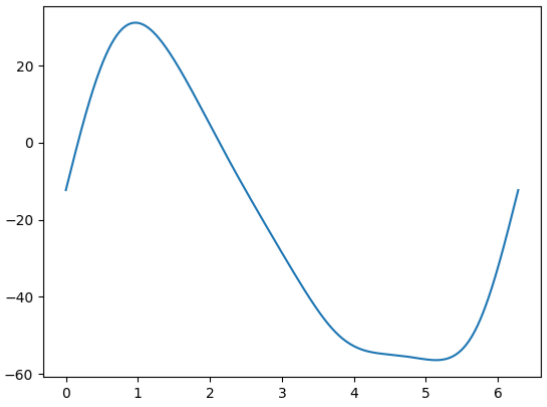
Firstly we will create a data object containing a surface $g : \mathbb{R}^2 \to \mathbb{R}$.

```
# Constructs example surface
X, Y, Z = axes3d.get_test_data(1.2)
data_matrix = [Z.T]
sample_points = [X[0,:], Y[:, 0]]

g = skfda.FDataGrid(data_matrix, sample_points)

# Sets cubic interpolation
g.interpolator =
skfda.representation.interpolation.SplineInterpolator(interpolation_order=3)

# Plots the surface
g.plot()
```



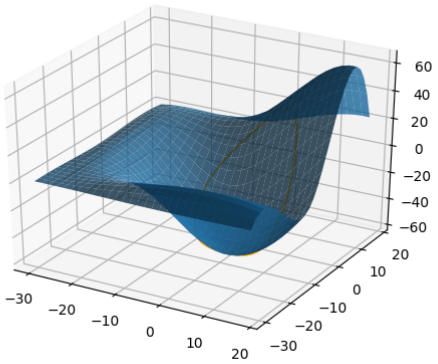
We will create a parametric curve $f(t) = (10\ \cos(t), 10\ sin(t))$. The result of the composition, $g \circ f : \mathbb{R} \to \mathbb{R}$ will be another functional object with the values of g along the path given by f.

```
# Creation of circunference in parametric form
t = np.linspace(0, 2*np.pi, 100)

data_matrix = [10 * np.array([np.cos(t), np.sin(t)]).T]
f = skfda.FDataGrid(data_matrix, t)

# Composition of function
gof = g.compose(f)

plt.figure()

gof.plot()
```



In the following chart it is plotted the curve $(10\ \cos(t), 10\ sin(t), g \circ f(t))$ and the surface.

```
# Plots surface
fig, ax = g.plot(alpha=.8)

# Plots path along the surface
path = f(t)[0]
ax[0].plot(path[:,0], path[:,1], gof(t)[0], color="orange")

plt.show()
```



[1] Function composition https://en.wikipedia.org/wiki/Function_composition.

Total running time of the script: (0 minutes 0.647 seconds)

ï€ Download Python source code: plot_composition.py

ï€ Download Jupyter notebook: plot_composition.ipynb

Shift Registration of basis

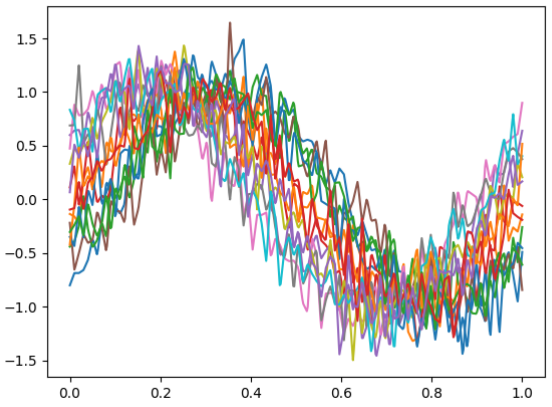
Shows the use of shift registration applied to a sinusoidal process represented in a Fourier basis.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

import skfda
import matplotlib.pyplot as plt
```

In this example we will use a `sinusoidal process` synthetically generated. This dataset consists in a sinusoidal wave with fixed period which contanis phase and amplitude variation with gaussian noise.

In this example we want to register the curves using a translation and remove the phase variation to perform further analysis.

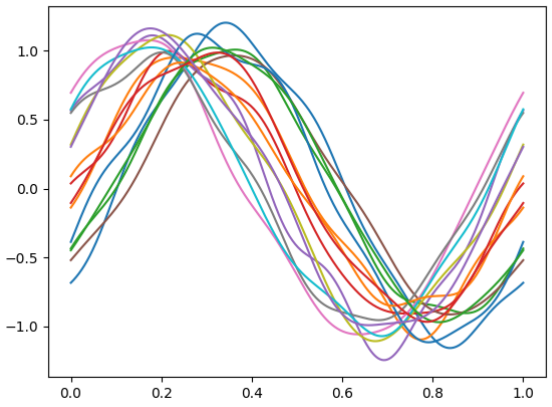
```
fd = skfda.datasets.make_sinusoidal_process(random_state=1)
fd.plot()
```



We will smooth the curves using a basis representation, which will help us to remove the gaussian noise. Smoothing before registration is essential due to the use of derivatives in the optimization process.

Because of their sinusoidal nature we will use a Fourier basis.

```
basis = skfda.representation.basis.Fourier(nbasis=11)
fd_basis = fd.to_basis(basis)

plt.figure()
fd_basis.plot()
```

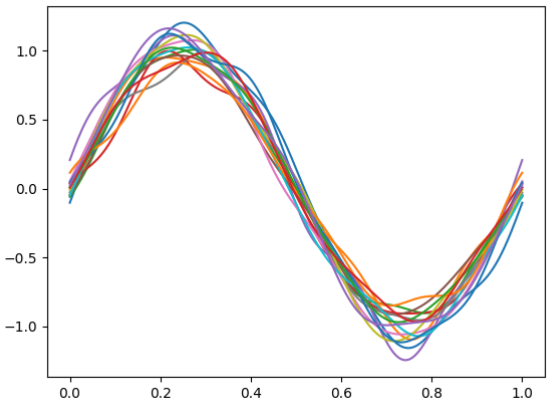


We will apply the `shift registration`, which is suitable due to the periodicity of the dataset and the small amount of amplitude variation.

```
fd_registered = skfda.preprocessing.registration.shift_registration(fd_basis)
```

We can observe how the sinusoidal pattern is easily distinguishable once the alignment has been made.

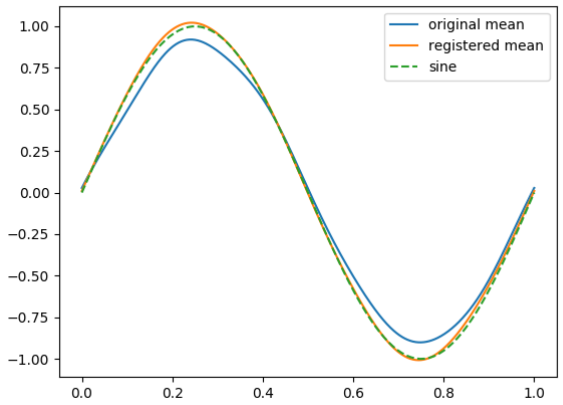
```
plt.figure()
fd_registered.plot()
```

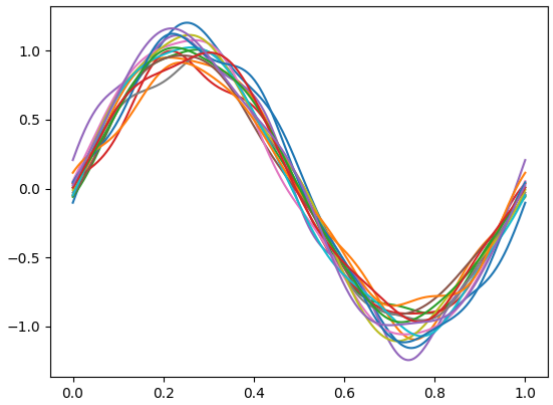


We will plot the mean of the original smoothed curves and the registered ones, and we will compare with the original sinusoidal process without noise.

We can see how the phase variation affects to the mean of the original curves varying their amplitude with respect to the original process, however, this effect is mitigated after the registration.

```
plt.figure()

fd_basis.mean().plot()
fd_registered.mean().plot()

# sinusoidal process without variation and noise
sine = skfda.datasets.make_sinusoidal_process(n_samples=1, phase_std=0,
                                              amplitude_std=0, error_std=0)

sine.plot(linestyle='dashed')

plt.legend(['original mean', 'registered mean','sine'])
```





The values of the shifts δ_i may be relevant for further analysis, as they may be considered as nuisance or random effects.

```
deltas = skfda.preprocessing.registration.shift_registration_deltas(fd_basis)
print(deltas)
```

Out:

```
[ 0.09004943  0.01808744  0.08732826 -0.00013559 -0.04950421  0.11984576
 -0.09723283 -0.09330286 -0.04398832 -0.08389279  0.0583045   0.00503724
  0.08788296  0.0214795  -0.042531  ]
```

The aligned functions can be obtained from the δ_i list using the *shift* method.

```
fd_basis.shift(deltas).plot()
```

Total running time of the script: (0 minutes 0.817 seconds)

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

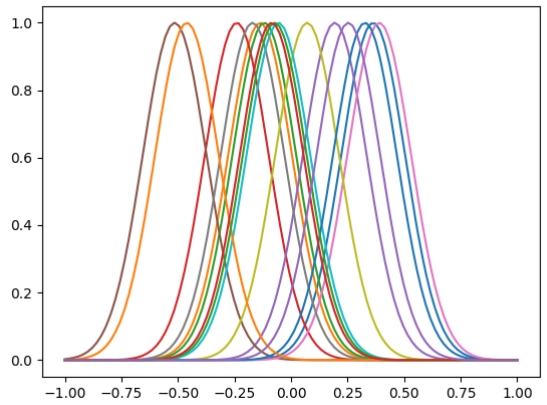
Landmark shift

This example shows how to shift functional data objects to align its samples with a particular reference point.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 2


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

We will use an example dataset synthetically generated by `make_multimodal_samples`, wich in this case will be used to generate gaussian-like samples with a mode near to 0. Each sample will be shifted to align their modes to a reference point using the function `landmark_shift`.

```
fd = skfda.datasets.make_multimodal_samples(random_state=1)
fd.extrapolation = 'bounds' # See extrapolation for a detailed explanation.

fd.plot()
```

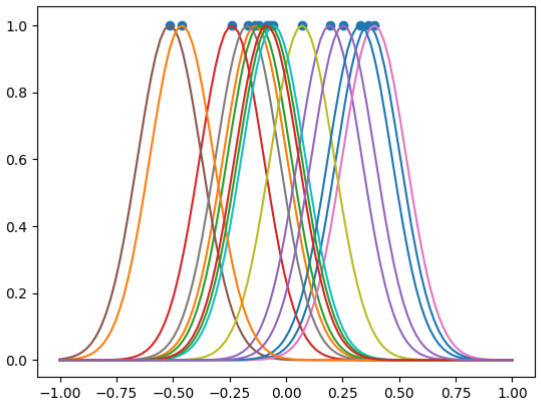


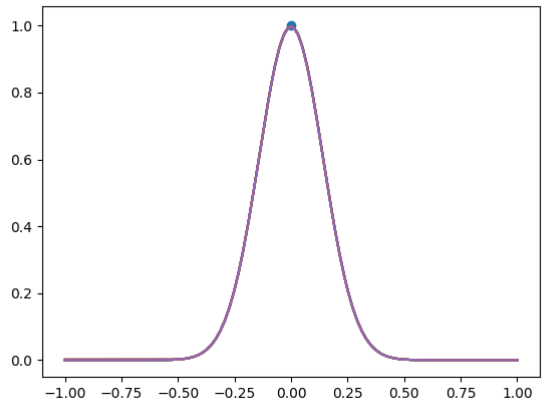
A landmark or a feature of a curve is some characteristic that one can associate with a specific argument value t. These are typically maxima, minima, or zero crossings of curves, and may be identified at the level of some derivatives as well as at the level of the curves themselves. [1]

For alignment we need to know in advance the location of the landmark of each of the samples, in our case it will correspond to the maxima of each sample. Because our dataset has been generated synthetically we can obtain the value of the landmarks using the function `make_multimodal_landmarks`, which is used by `make_multimodal_samples` to set the location of the modes.

In general it will be necessary to use numerical or other methods to determine the location of the landmarks.

```
landmarks = skfda.datasets.make_multimodal_landmarks(random_state=1).squeeze()

plt.figure()
plt.scatter(landmarks, np.repeat(1, fd.nsamples))
fd.plot()
```





Location of the landmarks:

```
print(landmarks)
```

Out:

```
[ 0.36321467 -0.13679289 -0.11810279 -0.23992308  0.19351103 -0.5146397
  0.39015177 -0.17021104  0.07133931 -0.05576091  0.32693727 -0.46066147
 -0.07209468 -0.08587716  0.25351855]
```

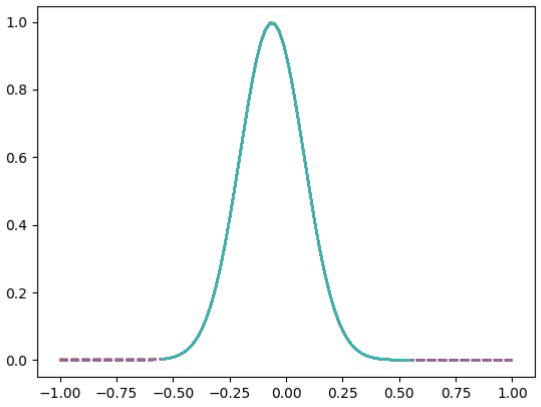
The following figure shows the result of shifting the curves to align their landmarks at 0.

```
fd_registered = skfda.preprocessing.registration.landmark_shift(fd, landmarks,
location=0)

plt.figure()
fd_registered.plot()
plt.scatter(0, 1)
```

In many circumstances it is possible that we could not apply extrapolation, in these cases it is possible to restrict the domain to avoid evaluating points outside where our curves are defined.

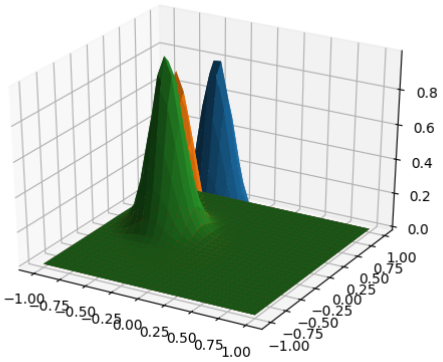
If the location of the new reference point is not specified it is choosen the point that minimizes the maximun amount of shift.

```
# Curves aligned restricting the domain
fd_restricted = skfda.preprocessing.registration.landmark_shift(fd, landmarks,
                                                                restrict_domain=True)

# Curves aligned to default point without restrict domain
fd_extrapolated = skfda.preprocessing.registration.landmark_shift(fd, landmarks)


plt.figure()
l1 = fd_extrapolated.plot(linestyle='dashed', label='Extrapolated samples')
l2 = fd_restricted.plot(label="Restricted samples")
#plt.legend(handles=[l1[-1], l2[-1]])
```



The previous method is also applicable for multidimensional objects, without limitation of the domain or image dimension. As an example we are going to create a datset with surfaces, in a similar way to the previous case.

```
fd = skfda.datasets.make_multimodal_samples(n_samples=3, points_per_dim=30,
                                            ndim_domain=2, random_state=1)

fd.plot()
```



In this case the landmarks will be defined by tuples with 2 coordinates.

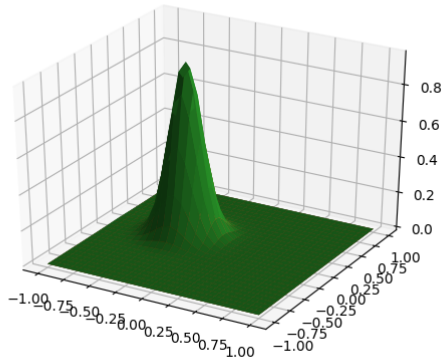
```
landmarks = skfda.datasets.make_multimodal_landmarks(n_samples=3, ndim_domain=2,
                                                     random_state=1).squeeze()
print(landmarks)
```

Out:

```
[[ 0.36321467 -0.13679289]
 [-0.11810279 -0.23992308]
 [ 0.19351103 -0.5146397 ]]
```

As in the previous case, we can align the curves to a specific point, or by default will be chosen the point that minimizes the maximum amount of displacement.

```
fd_registered = skfda.preprocessing.registration.landmark_shift(fd, landmarks)

fd_registered.plot()

plt.show()
```



[1] Ramsay, J., Silverman, B. W. (2005). Functional Data Analysis. Springer.

Total running time of the script: (0 minutes 1.899 seconds)

Download Python source code: plot_landmark_shift.py

Download Jupyter notebook: plot_landmark_shift.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

Landmark registration

This example shows the basic usage of the landmark registration.

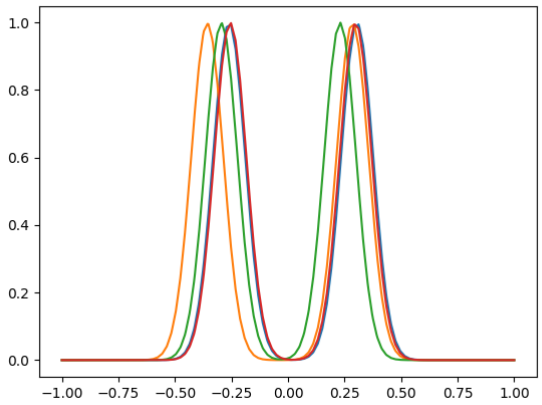
```
# Author: Pablo Marcos Manchón
# License: MIT

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

The simplest curve alignment procedure is landmark registration. This method only takes into account a discrete ammount of features of the curves which will be registered.

A landmark or a feature of a curve is some characteristic that one can associate with a specific argument value t. These are typically maxima, minima, or zero crossings of curves, and may be identified at the level of some derivatives as well as at the level of the curves themselves. We align the curves by transforming t for each curve so that landmark locations are the same for all curves. [1][2]

We will use a dataset synthetically generated by `make_multimodal_samples`, wich in this case will be used to generate bimodal curves.

```
fd = skfda.datasets.make_multimodal_samples(n_samples=4, n_modes=2, std=.002,
                                            mode_std=.005, random_state=1)
fd.plot()
```



For this type of alignment we need to know in advance the location of the landmarks of each of the samples, in our case it will correspond to the two maximun points of each sample. Because our dataset has been generated synthetically we can obtain the value of the landmarks using the function `make_multimodal_landmarks`, which is used by `make_multimodal_samples` to set the location of the modes.

In general it will be necessary to use numerical or other methods to determine the location of the landmarks.

```
landmarks = skfda.datasets.make_multimodal_landmarks(n_samples=4, n_modes=2,
                                                     std=.002, random_state=1
                                                     ).squeeze()

print(landmarks)
```

Out:

```
[[-0.2606904   0.30597475]
 [-0.35695389  0.28534872]
 [-0.29463113  0.23040539]
 [-0.25530298  0.29929113]]
```

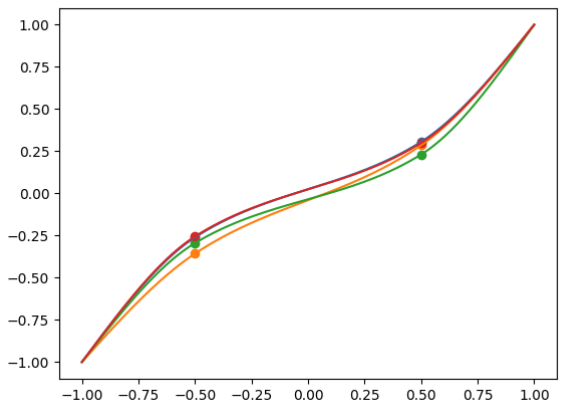
The transformation will not be linear, and will be the result of applying a warping function to the time of our curves.

After the identification of the landmarks asociated with the features of each of our curves we can construct the warping function with the function `landmark_registration_warping`.

Let h_i be the warping function corresponding with the curve i, t_{ij} the time where the curve i has their feature j and t_j^* the new location of the feature j. The warping functions will transform the new time in the original time of the curve, i.e., $h_i(t_j^*) = t_{ij}$. These functions will be defined between landmarks using monotone cubic interpolation (see the example of interpolation for more details).

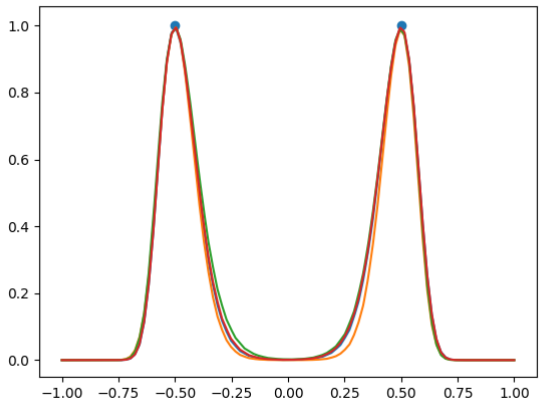
In this case we will place the landmarks at -0.5 and 0.5.

```
warping = skfda.preprocessing.registration.landmark_registration_warping(fd, landmarks,
                                                location=[-0.5, 0.5])

plt.figure()

# Plots warping
warping.plot()

# Plot landmarks
for i in range(fd.nsamples):
    plt.scatter([-0.5, 0.5], landmarks[i])
```



Once we have the warping functions, the registered curves can be obtained using function composition. Let x_i a curve, we can obtain the corresponding registered curve as $x_i^*(t) = x_i(h_i(t))$.

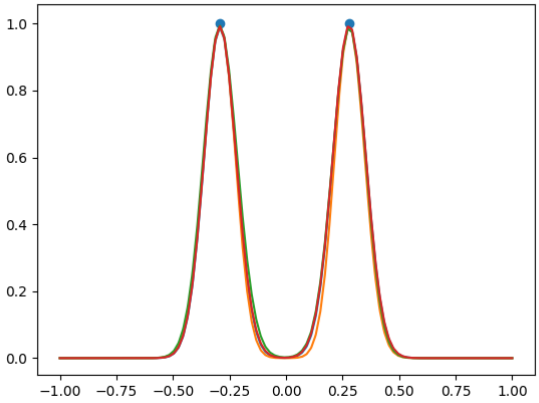
```
fd_registered = fd.compose(warping)
fd_registered.plot()

plt.scatter([-0.5, 0.5], [1, 1])
```



If we do not need the warping function we can obtain the registered curves directly using the function `landmark_registration`.

If the position of the new location of the landmarks is not specified the mean position is taken.

```
fd_registered = skfda.preprocessing.registration.landmark_registration(fd, landmarks)
fd_registered.plot()

plt.scatter(np.mean(landmarks, axis=0), [1, 1])


plt.show()
```



[1] Ramsay, J., Silverman, B. W. (2005). Functional Data Analysis. Springer.

[2] Ramsay, J., Hooker, G. & Graves S. (2009). Functional Data Analysis with R and Matlab. Springer.

Total running time of the script: (0 minutes 0.629 seconds)

Download Python source code: plot_landmark_registration.py

Download Jupyter notebook: plot_landmark_registration.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

Pairwise alignment

Shows the usage of the elastic registration to perform a pairwise alignment.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 5


import skfda
import matplotlib.pyplot as plt
import matplotlib.colors as clr
import numpy as np
```

Given any two functions f and g, we define their pairwise alignment or registration to be the problem of finding a warping function γ^* such that a certain energy term $E[f, g \circ \gamma]$ is minimized.

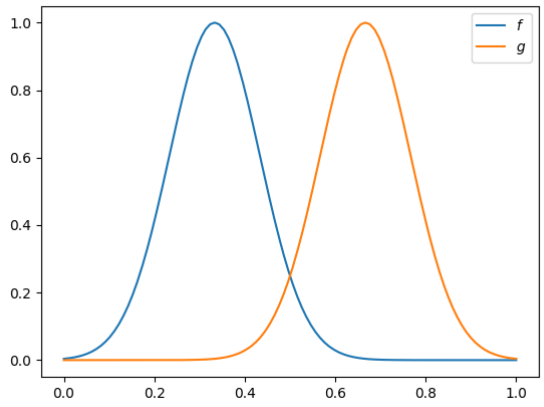
$$\gamma^* = *argmin_{\gamma \in \Gamma} E[f \circ \gamma, g]$$

In the case of elastic registration it is taken as energy function the Fisher-Rao distance with a penalisation term, due to the property of invariance to reparameterizations of warpings functions.

$$E[f \circ \gamma, g] = d_{FR}(f \circ \gamma, g)$$

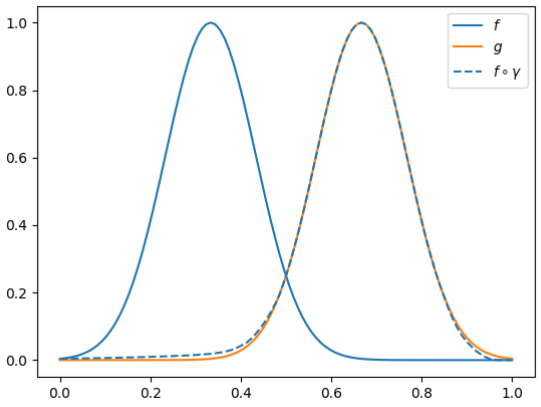
Firstly, we will create two unimodal samples, f and g, defined in [0, 1] wich will be used to show the elastic registration. Due to the similarity of these curves can be aligned almost perfectly between them.

```
# Samples with modes in 1/3 and 2/3
fd = skfda.datasets.make_multimodal_samples(n_samples=2, modes_location=[1/3,2/3],
                                            random_state=1, start=0, mode_std=.01)

fd.plot()
plt.legend(['$f$', '$g$'])
```



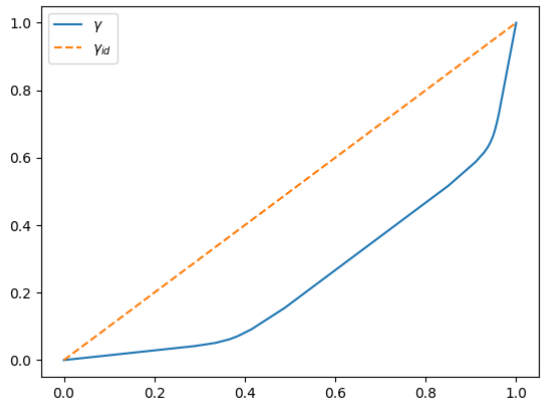
In this example g will be used as template and f will be aligned to it. In the following figure it is shown the result of the registration process, wich can be computed using `elastic_registration`.

```
f, g = fd[0], fd[1]

# Aligns f to g
fd_align = skfda.preprocessing.registration.elastic_registration(f, g)


plt.figure()

fd.plot()
fd_align.plot(color='C0', linestyle='--')


# Legend
plt.legend(['$f$', '$g$', '$f \\circ \\gamma $'])
```



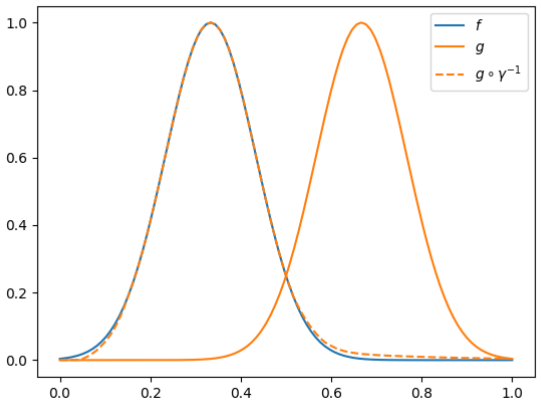
The non-linear transformation γ applied to f in the alignment can be obtained using `elastic_registration_warping`.

```
# Warping to align f to g
warping = skfda.preprocessing.registration.elastic_registration_warping(f, g)

plt.figure()

# Warping used
warping.plot()

# Plot identity
t = np.linspace(0, 1)
plt.plot(t, t, linestyle='--')

# Legend
plt.legend(['$\\gamma$', '$\\gamma_{id}$'])
```



The transformation necessary to align g to f will be the inverse of the original warping function, γ^{-1}. This fact is a consequence of the use of the Fisher-Rao metric as energy function.

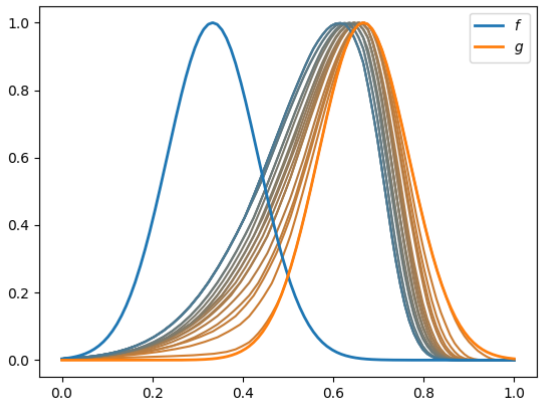
```
warping_inverse = skfda.preprocessing.registration.invert_warping(warping)


plt.figure()

fd.plot(label='$f$')
g.compose(warping_inverse).plot(color='C1', linestyle='--')


# Legend
plt.legend(['$f$', '$g$', '$g \\circ \\gamma^{-1} $'])
```



The amount of deformation used in the registration can be controlled by using a variation of the metric with a penalty term $\lambda \mathcal{R}(\gamma)$ wich will reduce the elasticity of the metric.

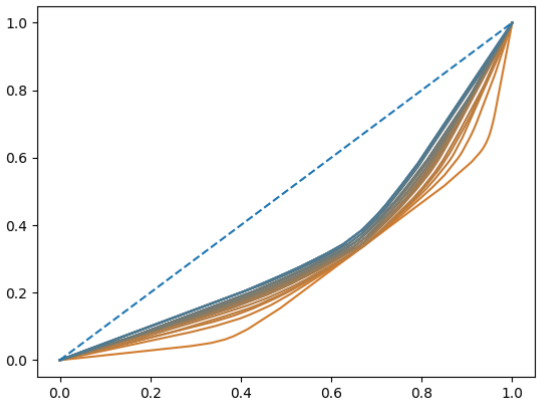
The following figure shows the original curves and the result to the alignment varying λ from 0 to 0.2.

```
# Values of lambda
lambdas = np.linspace(0, .2, 20)

# Creation of a color gradient
cmap = clr.LinearSegmentedColormap.from_list('custom cmap', ['C1','C0'])
color = cmap(.2 + 3*lambdas)

plt.figure()

for lam, c in zip(lambdas, color):
    # Plots result of alignment
    skfda.preprocessing.registration.elastic_registration(f, g, lam=lam).plot(color=c)


f.plot(color='C0', linewidth=2., label='$f$')
g.plot(color='C1', linewidth=2., label='$g$')

# Legend
plt.legend()
```



This phenomenon of loss of elasticity is clearly observed in the warpings used, since as the term of penalty increases, the functions are closer to γ_{id}.

```
plt.figure()

for lam, c in zip(lambdas, color):
    skfda.preprocessing.registration.elastic_registration_warping(f, g,
lam=lam).plot(color=c)

# Plots identity
plt.plot(t,t,  color='C0', linestyle="--")
```

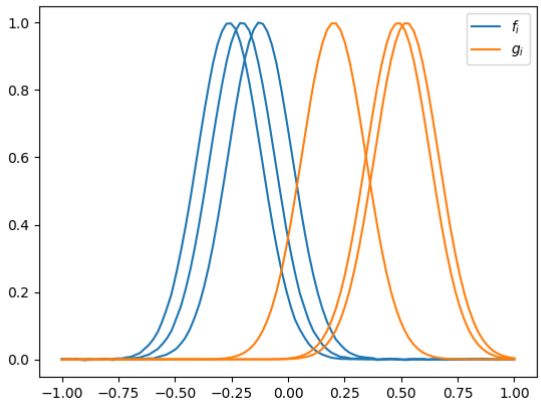


We can perform the pairwise of multiple curves at once. We can use a single curve as template to align a set of samples to it or a set of templates to make the alignemnt the two sets.

In the elastic registration example it is shown the alignment of multiple curves to the same template.

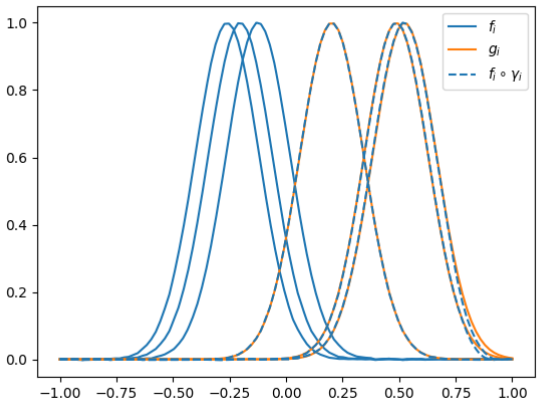
We will build two sets with 3 curves each, $\{f_i\}$ and $\{g_i\}$.

```
# Creation of the 2 sets of functions
state = np.random.RandomState(0)

location1 = state.normal(loc=-.3, scale=.1, size=3)
fd = skfda.datasets.make_multimodal_samples(n_samples=3, modes_location=location1,
                                            noise=.001 ,random_state=1)

location2 = state.normal(loc=.3, scale=.1, size=3)
g = skfda.datasets.make_multimodal_samples(n_samples=3, modes_location=location2,
                                           random_state=2)

# Plot of the sets
plt.figure()

fd.plot(color="C0", label="$f_i$")
fig, ax = g.plot(color="C1", label="$g_i$")

l = ax[0].get_lines()
plt.legend(handles=[l[0], l[-1]])
```



The following figure shows the result of the pairwise alignment of $\{f_i\}$ to $\{g_i\}$.

```
plt.figure()

# Registration of the sets
fd_registered = skfda.preprocessing.registration.elastic_registration(fd, g)

# Plot of the curves
fig, ax = fd.plot(color="C0", label="$f_i$")
l1 = ax[0].get_lines()[-1]
g.plot(color="C1", label="$g_i$")
l2 = ax[0].get_lines()[-1]
fd_registered.plot(color="C0", linestyle="--", label="$f_i \\circ \\gamma_i$")
l3 = ax[0].get_lines()[-1]

plt.legend(handles=[l1, l2, l3])

plt.show()
```



- Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Functional Data and Elastic Registration* (pp. 73-122). Springer.
- J. S. Marron, James O. Ramsay, Laura M. Sangalli and Anuj Srivastava (2015). Functional Data Analysis of Amplitude and Phase Variation. Statistical Science 2015, Vol. 30, No. 4

Total running time of the script: (0 minutes 2.988 seconds)

Download Python source code: plot_pairwise_alignment.py

Download Jupyter notebook: plot_pairwise_alignment.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

Elastic registration

Shows the usage of the elastic registration to perform a groupwise alignment.

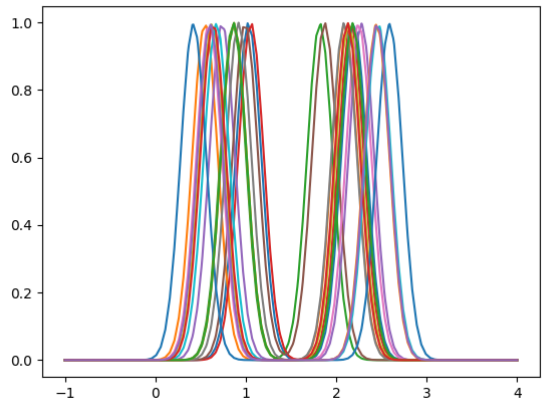
```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 5


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

In the example of pairwise alignment was shown the usage of `elastic_registration` to align a set of functional observations to a given template or a set of templates.

In the groupwise alignment all the samples are aligned to the same templated, constructed to minimise some distance, generally a mean or a median. In the case of the elastic registration, due to the use of the elastic distance in the alignment, one of the most suitable templates is the karcher mean under this metric.

We will create a synthetic dataset to show the basic usage of the registration.

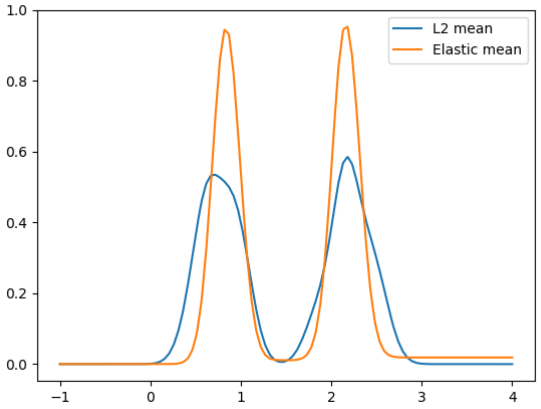
```
fd = skfda.datasets.make_multimodal_samples(n_modes=2, stop=4, random_state=1)
fd.plot()
```



The following figure shows the `elastic mean` of the dataset and the cross-sectional mean, which correspond to the karcher-mean under the $\mathbb{L}^2$ distance.

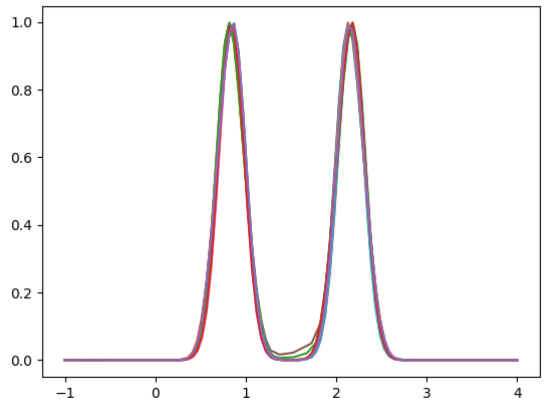
It can be seen how the elastic mean better captures the geometry of the curves compared to the standard mean, since it is not affected by the deformations of the curves.

```
plt.figure()
fd.mean().plot(label="L2 mean")
skfda.preprocessing.registration.elastic_mean(fd).plot(label="Elastic mean")
plt.legend()
```



In this case, the alignment completely reduces the amplitude variability between the samples, aligning the maximum points correctly.

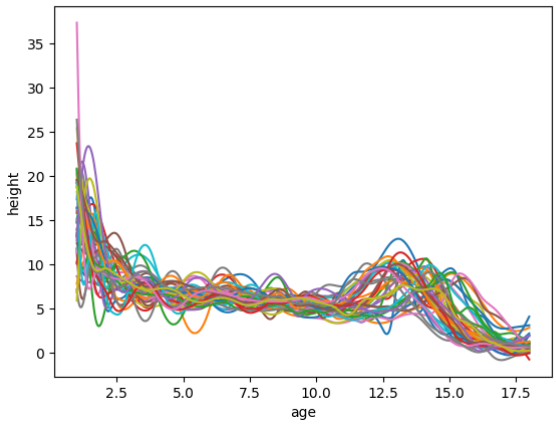
```
fd_align = skfda.preprocessing.registration.elastic_registration(fd)

plt.figure()
fd_align.plot()
```



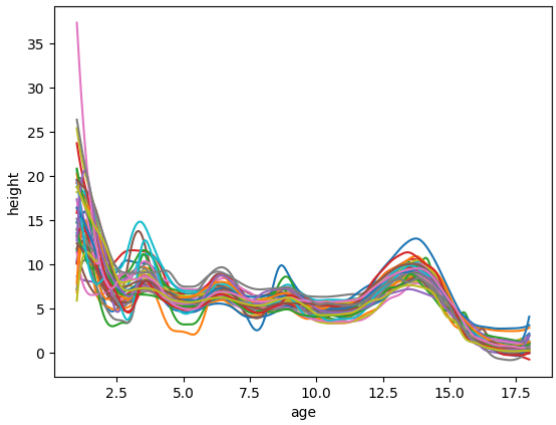
In general these type of alignments are not possible, in the following figure it is shown how it works with a real dataset. The `berkeley growth dataset` contains the growth curves of a set childs, in this case will be used only the males. The growth curves will be resampled using cubic interpolation and derived to obtain the velocity curves.

```
growth = skfda.datasets.fetch_growth()

# Select only one sex
fd = growth['data'][growth['target'] == 0]

# Obtain velocity curves
fd.interpolator = skfda.representation.interpolation.SplineInterpolator(3)
fd = fd.to_grid(np.linspace(*fd.domain_range[0], 200)).derivative()
fd = fd.to_grid(np.linspace(*fd.domain_range[0], 50))
fd.plot()

plt.figure()
fd_align = skfda.preprocessing.registration.elastic_registration(fd)
fd_align.dataset_label += " - aligned"

fd_align.plot()

plt.show()
```

Berkeley Growth Study - 1 derivative



Berkeley Growth Study - 1 derivative - aligned



- Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Functional Data and Elastic Registration* (pp. 73-122). Springer.
- J. S. Marron, James O. Ramsay, Laura M. Sangalli and Anuj Srivastava (2015). Functional Data Analysis of Amplitude and Phase Variation. Statistical Science 2015, Vol. 30, No. 4

Total running time of the script: (0 minutes 5.055 seconds)

Download Python source code: plot_elastic_registration.py

Download Jupyter notebook: plot_elastic_registration.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

K-nearest neighbors classification

Shows the usage of the k-nearest neighbors classifier.

```
# Author: Pablo Marcos Manchón
# License: MIT

import skfda
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from skfda.ml.classification import KNeighborsClassifier
```

In this example we are going to show the usage of the K-nearest neighbors classifier in their functional version, which is a extension of the multivariate one, but using functional metrics between the observations.

Firstly, we are going to fetch a functional data dataset, such as the Berkeley Growth Study. This dataset correspond to the height of several boys and girls measured until the 18 years of age.

We will try to predict the sex by using its growth curves.

The following figure shows the growth curves grouped by sex.

```
data = skfda.datasets.fetch_growth()
X = data['data']
y = data['target']

X[y==0].plot(color='C0')
X[y==1].plot(color='C1')
```

We will fit the classifier `KNeighborsClassifier` with the training partition. This classifier works exactly like the sklearn multivariate classifier `KNeighborsClassifier`, but will accept as input a `FDataGrid` with functional observations instead of an array with multivariate data.

```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
```

Once it is fitted, we can predict labels for the test samples.

To predict the label of a test sample, the classifier will calculate the k-nearest neighbors and will asign the majority class. By default, it is used the $\mathbb{L}^2$ distance between functions, to determine the neighbourhood of a sample, with 5 neighbors.

Can be used any of the functional metrics of the module `skfda.misc.metrics`.

```
pred = knn.predict(X_test)
print(pred)
```

Out:

```
[0 0 1 0 0 1 1 1 0 1 1 0 1 1 1 1 0 0 1 1 0 1 0 0 1 1 1 1 0 1 1]
```

The `score()` method allows us to calculate the mean accuracy for the test data. In this case we obtained around 96% of accuracy.

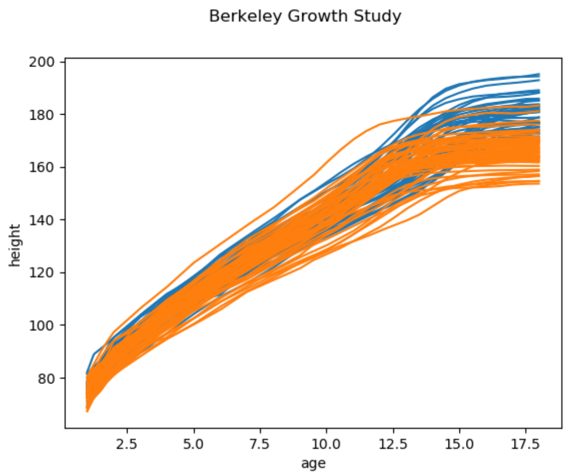
```
score = knn.score(X_test, y_test)
print(score)
```

Out:

```
0.967741935483871
```

We can also estimate the probability of membership to the predicted class using `predict_proba()`, which will return an array with the probabilities of the classes, in lexicographic order, for each test sample.

```
probs = knn.predict_proba(X_test[:5])
print(probs)
```



In this case, the class labels are stored in an array with 0's in the male samples and 1's in the positions with female ones.

```
print(y)
```

Out:

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
```

We can split the dataset using the sklearn function `train_test_split`.

We will use two thirds of the dataset for the training partition and the remaining samples for testing.

The function will return two `FDataGrid`'s, `X_train` and `X_test` with the corresponding partitions, and arrays with their class labels.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=0)
```

Out:

```
[[1.  0. ]
 [0.6 0.4]
 [0.  1. ]
 [0.6 0.4]
 [0.8 0.2]]
```

We can use the sklearn `GridSearchCV` to perform a grid search to select the best hyperparams, using cross-validation.

In this case, we will vary the number of neighbors between 1 and 11.

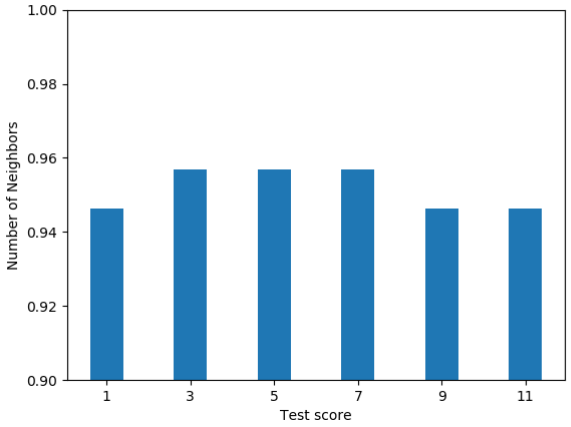
```
# only odd numbers
param_grid = {'n_neighbors': np.arange(1, 12, 2)}


knn = KNeighborsClassifier()
gscv = GridSearchCV(knn, param_grid, cv=KFold(shuffle=True, random_state=0))
gscv.fit(X, y)


print("Best params:", gscv.best_params_)
print("Best score:", gscv.best_score_)
```

Out:

```
Best params: {'n_neighbors': 3}
Best score: 0.956989247311828
```

We have obtained the greatest mean accuracy using 3 neighbors. The following figure shows the score depending on the number of neighbors.

```
plt.figure()
plt.bar(param_grid['n_neighbors'], gscv.cv_results_['mean_test_score'])

plt.xticks(param_grid['n_neighbors'])
plt.ylabel("Number of Neighbors")
plt.xlabel("Test score")
plt.ylim((0.9, 1))
```



In this dataset, the functional observations have been sampled equiespaciated. If we approximate the integral of the $\mathbb{L}^2$ distance as a Riemann sum (actually the Simpson's rule it is used), we obtain that it is approximately equivalent to the euclidean distance between vectors.

$$\|f - g\|_{\mathbb{L}^2} = \left(\int_a^b |f(x) - g(x)|^2 dx\right)^{\frac{1}{2}} \approx \left(\sum_{n=0}^{N} \triangle\, h\, |f(x_n) - g(x_n)|^2\right)^{\frac{1}{2}} = \sqrt{\triangle\, h}\, d_{euclidean}(\vec{f}, \vec{g})$$

So, in this case, it is roughtly equivalent to use this metric instead of the functional one, due to the constant multiplication do no affect the order of the neighbors.

Setting the parameter `sklearn_metric` of the classifier to True, a vectorial metric of sklearn can be passed. In `sklearn.neighbors.DistanceMetric` there are listed all the metrics supported.

We will fit the model with the sklearn distance and search for the best parameter. The results can vary sightly, due to the approximation during the integration, but the result should be similar.

```
knn = KNeighborsClassifier(metric='euclidean', sklearn_metric=True)
gscv2 = GridSearchCV(knn, param_grid, cv=KFold(shuffle=True, random_state=0))
gscv2.fit(X, y)

print("Best params:", gscv2.best_params_)
print("Best score:", gscv2.best_score_)
```

Out:

```
Best params: {'n_neighbors': 7}
Best score: 0.967741935483871
```

The advantage of use the sklearn metrics is the computational speed, three orders of magnitude faster. But it is not always possible to resample samples equiespaced nor do all functional metrics have a vector equivalent in this way.

The mean score time depending on the metric is shown below.

```
print("Mean score time (seconds)")
print("L2 distance:", np.mean(gscv.cv_results_['mean_score_time']), "(s)")
print("Sklearn distance:", np.mean(gscv2.cv_results_['mean_score_time']), "(s)")
```

Out:

```
Mean score time (seconds)
L2 distance: 0.609623114267985 (s)
Sklearn distance: 0.0009377532535129123 (s)
```

This classifier can be used with multivariate funcional data, as surfaces or curves in $\mathbb{R}^N$, if the metric support it too.

```
plt.show()
```

Total running time of the script: (0 minutes 34.861 seconds)

Download Python source code: plot_k_neighbors_classification.py

Download Jupyter notebook: plot_k_neighbors_classification.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

Radius neighbors classification

Shows the usage of the radius nearest neighbors classifier.

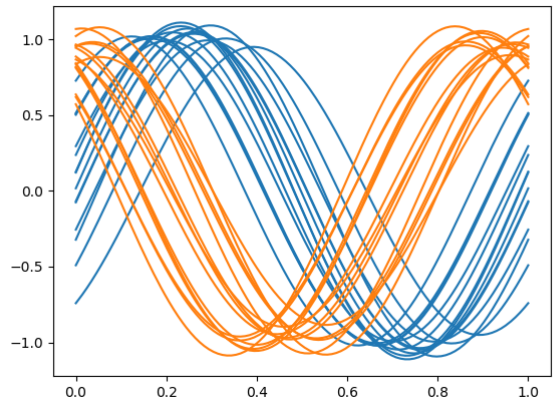
```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 2


import skfda
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from skfda.ml.classification import RadiusNeighborsClassifier
from skfda.misc.metrics import pairwise_distance, lp_distance
```

In this example, we are going to show the usage of the radius nearest neighbors classifier in their functional version, a variation of the K-nearest neighbors classifier, where it is used a vote among neighbors within a given radius, instead of use the k nearest neighbors.

Firstly, we will construct a toy dataset to show the basic usage of the API.

We will create two classes of sinusoidal samples, with different locations of their phase.

```
fd1 = skfda.datasets.make_sinusoidal_process(error_std=.0, phase_std=.35,
                                             random_state=0)
fd2 = skfda.datasets.make_sinusoidal_process(phase_mean=1.9, error_std=.0,
                                             random_state=1)

fd1.plot(color='C0')
fd2.plot(color='C1')
```



As in the K-nearest neighbor example, we will split the dataset in two partitions, for training and test, using the sklearn function `sklearn.model_selection.train_test_split()`.

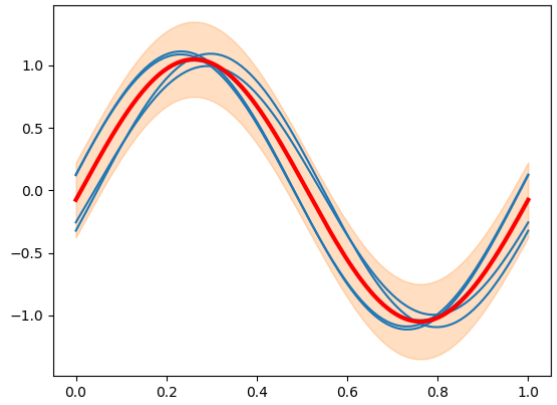
```
# Concatenate the two classes in the same FDataGrid
X = fd1.concatenate(fd2)
y = np.array(15*[0] + 15*[1])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
                                                    shuffle=True, random_state=0)
```

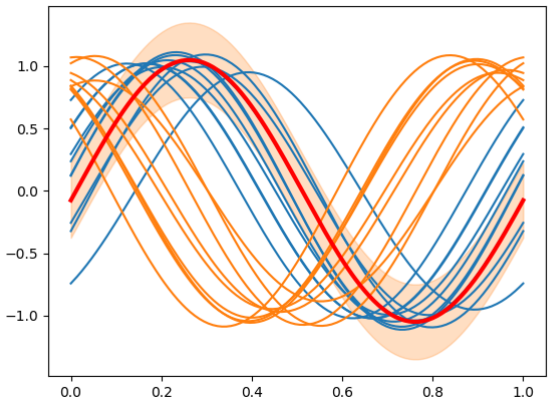
As in the multivariate data, the label assigned to a test sample will be the majority class of its neighbors, in this case all the samples in the ball center in the sample.

If we use the $\mathbb{L}^\infty$ metric, we can visualize a ball as a bandwidth with a fixed radius around a function.

The following figure shows the ball centered in the first sample of the test partition.

```
plt.figure()

sample = X_test[0]

X_train[y_train == 0].plot(color='C0')
X_train[y_train == 1].plot(color='C1')
sample.plot(color='red', linewidth=3)

lower = sample - 0.3
upper = sample + 0.3

plt.fill_between(sample.sample_points[0], lower.data_matrix.flatten(),
                 upper.data_matrix[0].flatten(),  alpha=.25, color='C1')
```





We will fit the classifier `RadiusNeighborsClassifier`, which has a similar API than the sklearn estimator `sklearn.neighbors.RadiusNeighborsClassifier` but accepting `FDataGrid` instead of arrays with multivariate data.

The vote of the neighbors can be weighted using the paramenter `weights`. In this case we will weight the vote inversely proportional to the distance.

```
radius_nn = RadiusNeighborsClassifier(radius=.3,  weights='distance')
radius_nn.fit(X_train, y_train)
```

In this case, all the neighbors in the ball belong to the first class, so this will be the class predicted.

We can predict labels for the test partition with `predict()`.

```
# Creation of pairwise distance
l_inf = pairwise_distance(lp_distance, p=np.inf)
distances = l_inf(sample, X_train)[0] # L_inf distances to 'sample'

plt.figure()

X_train[distances <= .3].plot(color='C0')
sample.plot(color='red', linewidth=3)

plt.fill_between(sample.sample_points[0], lower.data_matrix.flatten(),
                 upper.data_matrix[0].flatten(),  alpha=.25, color='C1')
```

```
pred = radius_nn.predict(X_test)
print(pred)
```

Out:

```
[0 1 0 0 1 1 1 0 1 1]
```

In this case, we get 100% accuracy, althouth, it is a toy dataset and it does not have much merit.

```
test_score = radius_nn.score(X_test, y_test)
print(test_score)
```

Out:

```
1.0
```

As in the K-nearest neighbor example, we can use a sklearn metric approximately equivalent to the functional $\mathbb{L}^2$ one, but computationally faster.

We saw that $\|f - g\|_{\mathbb{L}^2} \approx \sqrt{\triangle h}\, d_{euclidean}(\vec{f}, \vec{g})$ if the samples are equiespaced (or almost).

In the KNN case, the constant $\sqrt{\triangle h}$ does not matter, but in this case will affect the value of the radius, dividing by $\sqrt{\triangle h}$.

In this dataset $\triangle h = 0.001$, so, we have to multiply the radius by 10 to achieve the same result.

The computation using this metric it is 1000 times faster. See the K-neighbors classifier example and the API documentation to get detailled information.

We obtain 100% accuracy with this metric too.

```
radius_nn = RadiusNeighborsClassifier(radius=3, metric='euclidean',
                                      weights='distance', sklearn_metric=True)


radius_nn.fit(X_train, y_train)

test_score = radius_nn.score(X_test, y_test)
print(test_score)
```

Out:

```
1.0
```

If the radius is too small, it is possible to get samples with no neighbors. The classifier will raise and exception in this case.

```
radius_nn.set_params(radius=.5) # Radius 0.05 in the L2 distance
radius_nn.fit(X_train, y_train)

try:
    radius_nn.predict(X_test)
except ValueError as e:
    print(e)
```

Out:

```
No neighbors found for test samples [3, 4, 5, 6, 7, 8, 9], you can try using larger
radius, give a label for outliers, or consider removing them from your dataset.
```

A label to these oulier samples can be provided to avoid this problem.

```
radius_nn.set_params(outlier_label=2)
radius_nn.fit(X_train, y_train)
pred = radius_nn.predict(X_test)

print(pred)
```

Out:

```
[0 1 0 2 2 2 2 2 2 2]
```

This classifier can be used with multivariate funcional data, as surfaces or curves in $\mathbb{R}^N$, if the metric support it too.

```
plt.show()
```

Total running time of the script: (0 minutes 0.617 seconds)

Download Python source code: plot_radius_neighbors_classification.py

Download Jupyter notebook: plot_radius_neighbors_classification.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code

Neighbors Scalar Regression

Shows the usage of the nearest neighbors regressor with scalar response.

```
# Author: Pablo Marcos Manchón
# License: MIT

# sphinx_gallery_thumbnail_number = 3

import skfda
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from skfda.ml.regression import KNeighborsScalarRegressor
from skfda.misc.metrics import norm_lp
```

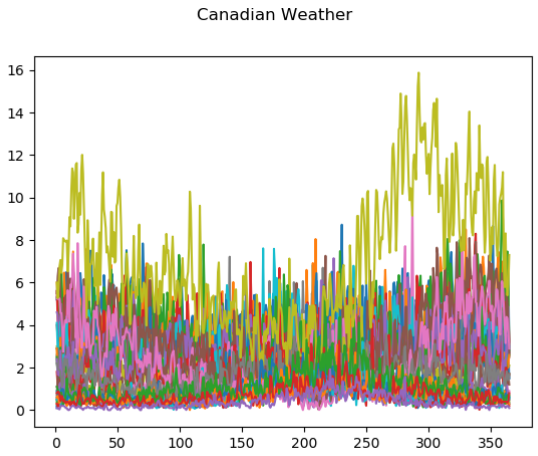
In this example, we are going to show the usage of the nearest neighbors regressors with scalar response. There is available a K-nn version, `KNeighborsScalarRegressor`, and other one based in the radius, `RadiusNeighborsScalarRegressor`.

Firstly we will fetch a dataset to show the basic usage.

The caniadian weather dataset contains the daily temperature and precipitation at 35 different locations in Canada averaged over 1960 to 1994.

The following figure shows the different temperature curves.

```
data = skfda.datasets.fetch_weather()
fd = data['data']

# TODO: Change this after merge operations-with-images
fd.axes_labels = None
X = fd.copy(data_matrix=fd.data_matrix[..., 0])


X.plot()
```



We will try to predict the total log precipitation, i.e, $logPrecTot_i = \log \int_0^{365} prec_i(t)dt$ using the temperature curves.

To obtain the precTot we will calculate the $\mathbb{L}^1$ norm of the precipitation curves.

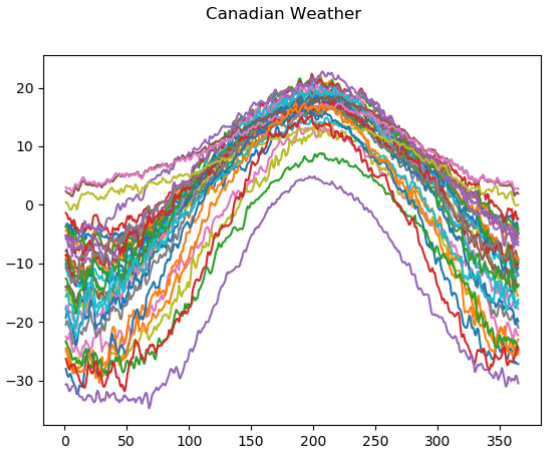
```
prec = norm_lp(y_func, 1)
log_prec = np.log(prec)

print(log_prec)
```

Out:

```
[7.29351642 7.276003   7.28963327 7.13669533 7.09071552 7.02467869
 6.6863602  6.8024318  6.83507705 7.0947603  7.00802285 6.83904808
 6.82668983 6.67653755 6.86835588 6.55881227 6.22620723 6.10709712
 6.01737593 5.90518018 6.00355779 5.89770317 6.14175063 6.00938632
 5.60138054 7.04457185  6.73308438 6.40539373 7.85460108 5.58537424
 5.78751069 5.58086126 6.01713229 5.56119432 4.96097809]
```

As in the nearest neighbors classifier examples, we will split the dataset in two partitions, for training and test, using the sklearn function `sklearn.model_selection.train_test_split()`.

```
X_train, X_test, y_train, y_test = train_test_split(X, log_prec, random_state=7)
```



In this example we are not interested in the precipitation curves directly, as in the case with regression response, we will train a nearest neighbor regressor to predict a scalar magnitude.

In the next figure the precipitation curves are shown.

```
y_func = fd.copy(data_matrix=fd.data_matrix[..., 1])

plt.figure()
y_func.plot()
```

Firstly we will try make a prediction with the default values of the estimator, using 5 neighbors and the $\mathbb{L}^2$.

We can fit the `KNeighborsScalarRegressor` in the same way than the sklearn estimators. This estimator is an extension of the sklearn `sklearn.neighbors.KNeighborsRegressor`, but accepting a `FDataGrid` as input instead of an array with multivariate data.

```
knn = KNeighborsScalarRegressor(weights='distance')
knn.fit(X_train, y_train)
```

We can predict values for the test partition using `predict()`.

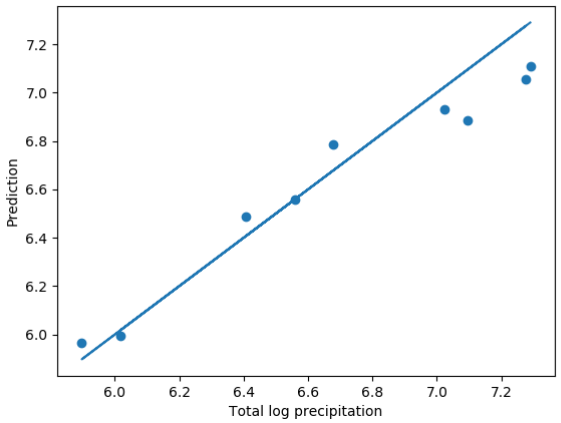
```
pred = knn.predict(X_test)
print(pred)
```

Out:

```
[7.10961556 5.99275367 7.05309816 6.88715827 6.78737106 5.96663073
 6.55900402 6.4855707  6.92975612]
```

The following figure compares the real precipitations with the predicted values.

```
plt.figure()
plt.scatter(y_test, pred)
plt.plot(y_test, y_test)
plt.xlabel("Total log precipitation")
plt.ylabel("Prediction")
```



We can quantify how much variability it is explained by the model with the coefficient of determination R^2 of the prediction, using `score()` for that.

The coefficient R^2 is defined as $(1 - u/v)$, where u is the residual sum of squares $\sum_i (y_i - y_{pred_i})^2$ and v is the total sum of squares $\sum_i (y_i - \bar{y})^2$.

```
score = knn.score(X_test, y_test)
print(score)
```

Out:

```
0.9266325148983648
```

In this case, we obtain a really good aproximation with this naive approach, although, due to the small number of samples, the results will depend on how the partition was done. In the above case, the explained variation is inflated for this reason.

We will perform cross-validation to test more robustly our model.

As in the neighbors classifiers examples, we can use a sklearn metric to approximate the $\mathbb{L}^2$ metric between function, but with a much lower computational cost.

Also, we can make a grid search, using `sklearn.model_selection.GridSearchCV`, to determine the optimal number of neighbors and the best way to weight their votes.

```
param_grid = {'n_neighbors': np.arange(1, 12, 2),
              'weights': ['uniform', 'distance']}


knn = KNeighborsScalarRegressor(metric='euclidean', sklearn_metric=True)
gscv = GridSearchCV(knn, param_grid, cv=KFold(shuffle=True, random_state=0))
gscv.fit(X, log_prec)
```

We obtain that 7 is the optimal number of neighbors, and a lower value of the R^2 coefficient, but much closer to the real one.

```
print(gscv.best_params_)
print(gscv.best_score_)
```

Out:

```
{'n_neighbors': 7, 'weights': 'distance'}
0.5307446923249634
```

More detailed information about the canadian weather dataset can be obtained in the following references.

- Ramsay, James O., and Silverman, Bernard W. (2006). Functional Data Analysis, 2nd ed. , Springer, New York.
- Ramsay, James O., and Silverman, Bernard W. (2002). Applied Functional Data Analysis, Springer, New Yorkn'


```
plt.show()
```

Total running time of the script: (0 minutes 0.923 seconds)

Download Python source code: plot_neighbors_scalar_regression.py

Download Jupyter notebook: plot_neighbors_scalar_regression.ipynb

Gallery generated by Sphinx-Gallery

C

Programmer's guide

In this annex may be found the documentation generated during this work, it is a sub-collection of the complete documentation available online.

Representation of functional Data

Before beginning to use the functionalities of the package, it is necessary to represent the data in functional form, using one of the following classes, which allow the visualization, evaluation and treatment of the data in a simple way, using the advantages of the object-oriented programming.

Discrete representation

A functional datum may be treated using a non-parametric representation, storing the values of the functions in a finite grid of points. The FDataGrid class supports multivariate functions using this approach. In the discretized function representation example it is shown the creation and basic visualisation of a FDataGrid.

`skfda.representation.grid.FDataGrid` (data_matrix)	Represent discretised functional data.

Functional data grids may be evaluated using interpolation, as it is shown in the interpolation example. The following class allows interpolation with different splines.

`skfda.representation.interpolation.SplineInterpolator` ([...])	Spline interpolator of `FDataGrid`.

Basis representation

The package supports a parametric representation using a linear combination of elements of a basis function system.

`skfda.representation.basis.FDataBasis` (basis, ...)	Basis representation of functional data.

The following classes are used to define different basis systems.

`skfda.representation.basis.BSpline` ([...])	BSpline basis.
`skfda.representation.basis.Fourier` ([...])	Fourier basis.
`skfda.representation.basis.Monomial` ([...])	Monomial basis.

Generic representation

Functional objects of the package are instances of FData, which contains the common attributes and methods used in all representations. This is an abstract class and cannot be instantiated directly, because it does not specify the representation of the data. Many of the package's functionalities receive an element of this class as an argument.

`skfda.representation.FData` (extrapolation, ...)	Defines the structure of a functional data object.

Extrapolation

All representations of functional data allow evaluation outside of the original interval using extrapolation methods.

- Extrapolation
  - Extrapolation Methods
    - skfda.representation.extrapolation.BoundaryExtrapolation
    - skfda.representation.extrapolation.ExceptionExtrapolation
    - skfda.representation.extrapolation.FillExtrapolation
    - skfda.representation.extrapolation.PeriodicExtrapolation
  - Custom Extrapolation
    - skfda.representation.evaluator.EvaluatorConstructor
    - skfda.representation.evaluator.Evaluator

skfda.representation.FData

class **skfda.representation.FData**(*extrapolation*, *dataset_label*, *axes_labels*, *keepdims*) [source]

Defines the structure of a functional data object.

nsamples

Number of samples.

Type: int

ndim_domain

Dimension of the domain.

Type: int

ndim_image

Dimension of the image.

Type: int

extrapolation

Default extrapolation mode.

Type: Extrapolation

dataset_label

name of the dataset.

Type: str

axes_labels

list containing the labels of the different axis. The first element is the x label, the second the y label and so on.

Type: list

keepdims

Default value of argument keepdims in `evaluate()`.

Type: bool

__init__(*extrapolation*, *dataset_label*, *axes_labels*, *keepdims*) [source]

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__` (extrapolation, dataset_label, ...)	Initialize self.
`argsort` ([ascending, kind])	Return the indices that would sort this array.
`astype` (dtype[, copy])	Cast to a NumPy array with 'dtype'.
`compose` (fd, *[, eval_points])	Composition of functions.
`concatenate` (other)	Join samples from a similar FData object.
`copy` (**kwargs)	Make a copy of the object.
`derivative` ([order])	Differentiate a FData object.
`dropna` ()	Return ExtensionArray without NA values
`evaluate` (eval_points, *[, derivative, ...])	Evaluate the object or its derivatives at a list
`factorize` ([na_sentinel])	Encode the extension array as an enumerate
`fillna` ([value, method, limit])	Fill NA/NaN values using the specified meth
`generic_plotting_checks` ([fig, ax, nrows, ncols])	Check the arguments passed to both `plot`
`isna` ()	A 1-D array indicating if each value is missin
`mean` ()	Compute the mean of all the samples.
`plot` ([chart, derivative, fig, ax, nrows, ...])	Plot the FDatGrid object.
`repeat` (repeats[, axis])	Repeat elements of a ExtensionArray.
`searchsorted` (value[, side, sorter])	Find indices where elements should be inser
`set_figure_and_axes` (nrows, ncols)	Set figure and its axes.
`set_labels` ([fig, ax, patches])	Set labels if any.

`shift` (shifts, *[, restrict_domain, ...])	Perform a shift of the curves.
`take` (indices[, allow_fill, fill_value])	Take elements from an array.
`to_basis` (basis[, eval_points])	Return the basis representation of the object
`to_grid` ([eval_points])	Return the discrete representation of the obj
`to_numpy` ()	Returns a numpy array with the objects
`unique` ()	Compute the ExtensionArray of unique value

Attributes

`domain_range`	Return the domain range of the object
`dtype`	An instance of 'ExtensionDtype'.
`extrapolation`	Return default type of extrapolation.
`extrapolator_evaluator`	Return the evaluator constructed by the extrapolator.
`nbytes`	The number of bytes needed to store this object in memory.
`ndim`	Return number of dimensions of the functional data.
`ndim_codomain`	Return number of dimensions of the codomain.
`ndim_domain`	Return number of dimensions of the domain.
`ndim_image`	Return number of dimensions of the image.
`nsamples`	Return the number of samples.
`shape`	Return a tuple of the array dimensions.

skfda.representation.interpolation.SplineInterpolator

class **skfda.representation.interpolation.SplineInterpolator**(*interpolation_order=1, smoothness_parameter=0.0, monotone=False*) [source]

Spline interpolator of `FDataGrid` .

Spline interpolator of discretized functional objects. Implements different interpolation methods based in splines, using the sample points of the grid as nodes to interpolate.

See the interpolation example to a detailled explanation.

`interpolator_order`

Order of the interpolation, 1 for linear interpolation, 2 for cuadratic, 3 for cubic and so on. In case of curves and surfaces there is available interpolation up to degree 5. For higher dimensional objects only linear or nearest interpolation is available. Default lineal interpolation.

Type: int, optional

`smoothness_parameter`

Penalisation to perform smoothness interpolation. Option only available for curves and surfaces. If 0 the residuals of the interpolation will be 0. Defaults 0.

Type: float, optional

`monotone`

Performs monotone interpolation in curves using a PCHIP interpolator. Only valid for curves (domain dimension equal to 1) and interpolation order equal to 1 or 3. Defaults false.

Type: boolean, optional

`__init__`(*interpolation_order=1, smoothness_parameter=0.0, monotone=False*) [source]

Constructor of the SplineInterpolator.

Parameters:

- **interpolator_order** (*int, optional*) – Order of the interpolation, 1 for linear interpolation, 2 for cuadratic, 3 for cubic and so on. In case of curves and surfaces there is available interpolation up to degree 5. For higher dimensional objects only linear or nearest interpolation is available. Default lineal interpolation.
- **smoothness_parameter** (*float, optional*) – Penalisation to perform smoothness interpolation. Option only available for curves and surfaces. If 0 the residuals of the interpolation will be 0. Defaults 0.
- **monotone** (*boolean, optional*) – Performs monotone interpolation in curves using a PCHIP interpolator. Only valid for curves (domain dimension equal to 1) and interpolation order equal to 1 or 3. Defaults false.

Methods

`__init__` ([interpolation_order, ...])	Constructor of the SplineInterpolator.
`evaluator` (fdatagrid)	Construct a SplineInterpolatorEvaluator used in the evalu

Attributes

`interpolation_order`	Returns the interpolation order
`monotone`	Returns flag to perform monotone interpolation
`smoothness_parameter`	Returns the smoothness parameter

Extrapolation

This module contains the extrapolators used to evaluate points outside the domain range of `FDataBasis` or `FDataGrid` . See Extrapolation Example for detailed explanation.

Extrapolation Methods

The following classes are used to define common methods of extrapolation.

`skfda.representation.extrapolation.BoundaryExtrapolation`	Extends the domain range using th
`skfda.representation.extrapolation.ExceptionExtrapolation`	Raise and exception.
`skfda.representation.extrapolation.FillExtrapolation` (...)	Values outside the domain range w
`skfda.representation.extrapolation.PeriodicExtrapolation`	Extends the domain range periodic

Custom Extrapolation

Custom extrapolators could be done subclassing `EvaluatorConstructor` .

`skfda.representation.evaluator.EvaluatorConstructor`	Constructor of an evaluator.
`skfda.representation.evaluator.Evaluator`	Structure of an evaluator.

skfda.representation.extrapolation.ExceptionExtrapolation

class **skfda.representation.extrapolation.ExceptionExtrapolation** [source]

Raise and exception.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import ExceptionExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = ExceptionExtrapolation()
>>> try:
...     fd([-.5, 0, 1.5]).round(3)
... except ValueError as e:
...     print(e)
Attempt to evaluate 2 points outside the domain range.
```

This extrapolator is equivalent to the string *"exception"*.

```
>>> fd.extrapolation = 'exception'
>>> try:
...     fd([-.5, 0, 1.5]).round(3)
... except ValueError as e:
...     print(e)
Attempt to evaluate 2 points outside the domain range.
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`evaluator`(fdata)	Returns the evaluator used by `FData`.

skfda.representation.extrapolation.PeriodicExtrapolation

class **skfda.representation.extrapolation.PeriodicExtrapolation** [source]

Extends the domain range periodically.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import PeriodicExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = PeriodicExtrapolation()
>>> fd([-.5, 0, 1.5]).round(3)
array([[-0.724,  0.976, -0.724],
       [-1.086,  0.759, -1.086]])
```

This extrapolator is equivalent to the string *"periodic"*

```
>>> fd.extrapolation = 'periodic'
>>> fd([-.5, 0, 1.5]).round(3)
array([[-0.724,  0.976, -0.724],
       [-1.086,  0.759, -1.086]])
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`evaluator`(fdata)	Returns the evaluator used by `FData`.

skfda.representation.extrapolation.BoundaryExtrapolation

class **skfda.representation.extrapolation.BoundaryExtrapolation** [source]

Extends the domain range using the boundary values.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import BoundaryExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = BoundaryExtrapolation()
>>> fd([-.5, 0, 1.5]).round(3)
array([[ 0.976,  0.976,  0.797],
       [ 0.759,  0.759,  1.125]])
```

This extrapolator is equivalent to the string *"bounds"*.

```
>>> fd.extrapolation = 'bounds'
>>> fd([-.5, 0, 1.5]).round(3)
array([[ 0.976,  0.976,  0.797],
       [ 0.759,  0.759,  1.125]])
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`evaluator`(fdata)	Returns the evaluator used by `FData`.

skfda.representation.extrapolation.FillExtrapolation

class **skfda.representation.extrapolation.FillExtrapolation**(*fill_value*) [source]

Values outside the domain range will be filled with a fixed value.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.extrapolation import FillExtrapolation
>>> fd = make_sinusoidal_process(n_samples=2, random_state=0)
```

We can set the default type of extrapolation

```
>>> fd.extrapolation = FillExtrapolation(0)
>>> fd([-.5, 0, 1.5]).round(3)
array([[ 0.   ,  0.976,  0.   ],
       [ 0.   ,  0.759,  0.   ]])
```

The previous extrapolator is equivalent to the string *"zeros"*. In the same way FillExtrapolation(np.nan) is equivalent to *"nan"*.

```
>>> fd.extrapolation = "nan"
>>> fd([-.5, 0, 1.5]).round(3)
array([[   nan,  0.976,    nan],
       [   nan,  0.759,    nan]])
```

__init__(*fill_value*) [source]

Returns the evaluator used by `FData`.

Returns: Evaluator of the periodic extrapolation.

Return type: (`Evaluator`)

Methods

`__init__`(fill_value)	Returns the evaluator used by `FData`.

`evaluator` (fdata)	Construct an evaluator.

Attributes

`fill_value`	Returns the fill value of the extrapolation

The elastic registration is a novel approach to this problem that uses the properties of the Fisher-Rao metric to perform the alignment of the curves. In the examples of pairwise alignment and elastic registration is shown a brief introduction to this topic along the usage of the corresponding functions.

`skfda.preprocessing.registration.elastic_registration` (...)	Align a FDatagrid using the
`skfda.preprocessing.registration.elastic_registration_warping` (...)	Calculate the warping to alig

The module contains some routines related with the elastic registration, making a transformation of the sampling, computing different means or distances based on the elastic framework.

`skfda.preprocessing.registration.elastic_mean` (...)	Compute the karcher mean under the elastic
`skfda.preprocessing.registration.warping_mean` (...)	Compute the karcher mean of a set of warpi
`skfda.preprocessing.registration.to_srsf` (...)	Calculate the square-root slope function (SR
`skfda.preprocessing.registration.from_srsf` (...)	Given a SRSF calculate the corresponding fu

Amplitude and Phase Decomposition

The amplitude and phase variation may be quantified by comparing a sample before and after registration. The package contains an implementation of the decomposition procedure developed by *Kneip and Ramsay (2008)*.

`skfda.preprocessing.registration.mse_decomposition` (...)	Compute mean square error measures f

Utility functions

There are some other method related with the registration problem in this module.

`skfda.preprocessing.registration.invert_warping` (...)	Compute the inverse of a diffeom
`skfda.preprocessing.registration.normalize_warping` (warping)	Rescale a warping to normalize th

References

- Ramsay, J., Silverman, B. W. (2005). Functional Data Analysis. Springer.
- Kneip, Alois & Ramsay, James. (2008). Quantifying amplitude and phase variation. Journal of the American Statistical Association.
- Ramsay, J., Hooker, G. & Graves S. (2009). Functional Data Analysis with R and Matlab. Springer.
- Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. Springer.

Registration

We see often that variation in functional observations involves phase and amplitude variation, which may hinder further analysis. That problem is treated during the registration process. This module contains procedures for the registration of the data.

Shift Registration

Many of the issues involved in registration can be solved by considering the simplest case, a simple shift in the time scale. This often happens because the time at which the recording process begins is arbitrary, and is unrelated to the beginning of the interesting segment of the data. In the Shift Registration Example it is shown the basic usage of this methods applied to periodic data.

`skfda.preprocessing.registration.shift_registration` (fd, *)	Perform shift registration of
`skfda.preprocessing.registration.shift_registration_deltas` (fd, *)	Return the lists of shifts used

Landmark Registration

Landmark registration aligns features applying a transformation of the time that takes all the times of a given feature into a common value.

The simplest case in which each sample presents a unique landmark can be solved by performing a translation in the time scale. See the Landmark Shift Example.

`skfda.preprocessing.registration.landmark_shift` (fd, ...)	Perform a shift of the curves to
`skfda.preprocessing.registration.landmark_shift_deltas` (fd, ...)	Returns the corresponding shifts

The general case of landmark registration may present multiple landmarks for each sample and a non-linear transformation in the time scale should be applied. See the Landmark Registration Example

`skfda.preprocessing.registration.landmark_registration` (fd, ...)	Perform landmark regist
`skfda.preprocessing.registration.landmark_registration_warping` (fd, ...)	Calculate the transform

Elastic Registration

- J. S. Marron, James O. Ramsay, Laura M. Sangalli and Anuj Srivastava (2015). Functional Data Analysis of Amplitude and Phase Variation. Statistical Science 2015, Vol. 30, No. 4

skfda.preprocessing.registration.shift_registration

skfda.preprocessing.registration.shift_registration(*fd*, ***, *maxiter=5*, *tol=0.01*, *restrict_domain=False*, *extrapolation=None*, *step_size=1*, *initial=None*, *eval_points=None*, ***kwargs*) [source]

Perform shift registration of the curves.

Realizes a registration of the curves, using shift aligment, as is defined in [RS05-7-2]. Calculates δ_i for each sample such that $x_i(t + \delta_i)$ minimizes the least squares criterion:

$$\text{REGSSE} = \sum_{i=1}^{N} \int_{\mathcal{T}} [x_i(t + \delta_i) - \hat{\mu}(t)]^2 ds$$

Estimates the shift parameter δ_i iteratively by using a modified Newton-Raphson algorithm, updating the mean in each iteration, as is described in detail in [RS05-7-9-1].

Parameters:

- **fd** (`FData`) – Functional data object to be registered.
- **maxiter** (*int*, *optional*) – Maximun number of iterations. Defaults to 5.
- **tol** (*float*, *optional*) – Tolerance allowable. The process will stop if $\max_i |\delta_i^{(\iota)} - \delta_i^{(\iota-1)}| < tol$. Default sets to 1e-2.
- **restrict_domain** (*bool*, *optional*) – If True restricts the domain to avoid evaluate points outside the domain using extrapolation. Defaults uses extrapolation.
- **extrapolation** (str or `Extrapolation` , optional) – Controls the extrapolation mode for elements outside the domain range. By default uses the method defined in fd. See :module: *extrapolation* to obtain more information.
- **step_size** (*int or float*, *optional*) – Parameter to adjust the rate of convergence in the Newton-Raphson algorithm, see [RS05-7-9-1]. Defaults to 1.
- **initial** (*array_like*, *optional*) – Initial estimation of shifts. Default uses a list of zeros for the initial shifts.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain the discrete representation of the object to integrate. If None is passed it calls numpy.linspace in FDataBasis and uses the *sample_points* in FDataGrids.
- ****kwargs** – Keyword arguments to be passed to `shift()` .

Returns: `FData` A `FData` object with the curves registered.

Raises: `ValueError` – If the initial array has different length than the number of samples.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.basis import Fourier
>>> from skfda.preprocessing.registration import shift_registration
>>> fd = make_sinusoidal_process(n_samples=2, error_std=0, random_state=1)
```

Registration of data in discretized form:

```
>>> shift_registration(fd)
FDataGrid(...)
```

Registration of data in basis form:

```
>>> fd = fd.to_basis(Fourier())
>>> shift_registration(fd)
FDataBasis(...)
```

References

[RS05-7-2] Ramsay, J., Silverman, B. W. (2005). Shift registration. In *Functional Data Analysis* (pp. 129-132). Springer.

[RS05-7-9-1] (*1*, *2*) Ramsay, J., Silverman, B. W. (2005). Shift registration by the Newton-Raphson algorithm. In *Functional Data Analysis* (pp. 142-144). Springer.

skfda.preprocessing.registration.shift_registration_deltas

skfda.preprocessing.registration.shift_registration_deltas(*fd*, ***, *maxiter=5*, *tol=0.01*, *restrict_domain=False*, *extrapolation=None*, *step_size=1*, *initial=None*, *eval_points=None*) [source]

Return the lists of shifts used in the shift registration procedure.

Realizes a registration of the curves, using shift aligment, as is defined in [RS05-7-2-1]. Calculates δ_i for each sample such that $x_i(t + \delta_i)$ minimizes the least squares criterion:

$$\text{REGSSE} = \sum_{i=1}^{N} \int_{\mathcal{T}} [x_i(t + \delta_i) - \hat{\mu}(t)]^2 ds$$

Estimates the shift parameter δ_i iteratively by using a modified Newton-Raphson algorithm, updating the mean in each iteration, as is described in detail in [RS05-7-9-1-1].

Method only implemented for Funtional objects with domain and image dimension equal to 1.

Parameters:

- **fd** (`FData`) – Functional data object to be registered.
- **maxiter** (*int*, *optional*) – Maximun number of iterations. Defaults to 5.
- **tol** (*float*, *optional*) – Tolerance allowable. The process will stop if $\max_i |\delta_i^{(\iota)} - \delta_i^{(\iota-1)}| < tol$. Default sets to 1e-2.
- **restrict_domain** (*bool*, *optional*) – If True restricts the domain to avoid evaluate points outside the domain using extrapolation. Defaults uses extrapolation.
- **extrapolation** (str or `Extrapolation` , optional) – Controls the extrapolation mode for elements outside the domain range. By default uses the method defined in fd. See :module: *extrapolation* to obtain more information.
- **step_size** (*int or float*, *optional*) – Parameter to adjust the rate of convergence in the Newton-Raphson algorithm, see [RS05-7-9-1-1]. Defaults to 1.
- **initial** (*array_like*, *optional*) – Initial estimation of shifts. Default uses a list of zeros for the initial shifts.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain the discrete representation of the object to integrate. If None is passed it calls numpy.linspace in FDataBasis and uses the *sample_points* in FDataGrids.

Returns: list with the shifts.

Return type: `numpy.ndarray`

Raises: `ValueError` – If the initial array has different length than the number of samples.

Examples

```
>>> from skfda.datasets import make_sinusoidal_process
>>> from skfda.representation.basis import Fourier
>>> from skfda.preprocessing.registration import shift_registration_deltas
>>> fd = make_sinusoidal_process(n_samples=2, error_std=0, random_state=1)
```

Registration of data in discretized form:

```
>>> shift_registration_deltas(fd).round(3)
array([-0.022,  0.03 ])
```

Registration of data in basis form:

```
>>> fd = fd.to_basis(Fourier())
>>> shift_registration_deltas(fd).round(3)
array([-0.022,  0.03 ])
```

References

[RS05-7-2-1] Ramsay, J., Silverman, B. W. (2005). Shift registration. In *Functional Data Analysis* (pp. 129-132). Springer.

[RS05-7-9-1-1] (*1*, *2*) Ramsay, J., Silverman, B. W. (2005). Shift registration by the Newton-Raphson algorithm. In *Functional Data Analysis* (pp. 142-144). Springer.


```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_shift
```

We will create a data with landmarks as example

```
>>> fd = make_multimodal_samples(n_samples=3, random_state=1)
>>> landmarks = make_multimodal_landmarks(n_samples=3, random_state=1)
>>> landmarks = landmarks.squeeze()
```

The function will return the sample registered

```
>>> landmark_shift(fd, landmarks)
FDataGrid(...)
```


skfda.preprocessing.registration.landmark_shift

skfda.preprocessing.registration.landmark_shift(*fd*, *landmarks*, *location=None*, ***, *restrict_domain=False*, *extrapolation=None*, *eval_points=None*, ***kwargs*) [source]

Perform a shift of the curves to align the landmarks.

Let t^* the time where the landmarks of the curves will be aligned, t_i the location of the landmarks for each curve and $\delta_i = t_i - t^*$.

The registered samples will have their feature aligned.

$$x_i^*(t^*) = x_i(t^* + \delta_i) = x_i(t_i)$$

Parameters:

- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List with the landmarks of the samples.
- **location** (*numeric or callable, optional*) – Defines where the landmarks will be alligned. If a numeric value is passed the landmarks will be alligned to it. In case of a callable is passed the location will be the result of the the call, the function should be accept as an unique parameter a numpy array with the list of landmarks. By default it will be used as location $\frac{1}{2}(max(\text{landmarks}) + min(\text{landmarks}))$ wich minimizes the max shift.
- **restrict_domain** (*bool, optional*) – If True restricts the domain to avoid evaluate points outside the domain using extrapolation. Defaults uses extrapolation.
- **extrapolation** (*str or Extrapolation, optional*) – Controls the extrapolation mode for elements outside the domain range. By default uses the method defined in fd. See extrapolation to more information.
- **eval_points** (*array_like, optional*) – Set of points where the functions are evaluated in `shift()` .
- ****kwargs** – Keyword arguments to be passed to `shift()` .

Returns: Functional data object with the registered samples.

Return type: `FData`

Examples

skfda.preprocessing.registration.landmark_shift_deltas

skfda.preprocessing.registration.landmark_shift_deltas(*fd*, *landmarks*, *location=None*) [source]

Returns the corresponding shifts to align the landmarks of the curves.

Let t^* the time where the landmarks of the curves will be aligned, and t_i the location of the landmarks for each curve. The function will calculate the corresponding δ_i shuch that $t_i = t^* + \delta_i$.

This procedure will work independent of the dimension of the domain and the image.

Parameters:

- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List with the landmarks of the samples.
- **location** (*numeric or callable, optional*) – Defines where the landmarks will be alligned. If a numer or list is passed the landmarks will be alligned to it. In case of a callable is passed the location will be the result of the the call, the function should be accept as an unique parameter a numpy array with the list of landmarks. By default it will be used as location $\frac{1}{2}(max(\text{landmarks}) + min(\text{landmarks}))$ wich minimizes the max shift.

Returns: Array containing the corresponding shifts.

Return type: `numpy.ndarray`

Raises: `ValueError` – If the list of landmarks does not match with the number of samples.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_shift_deltas
```

We will create a data with landmarks as example

```
>>> fd = make_multimodal_samples(n_samples=3, random_state=1)
>>> landmarks = make_multimodal_landmarks(n_samples=3, random_state=1)
>>> landmarks = landmarks.squeeze()
```

The function will return the corresponding shifts

```
>>> shifts = landmark_shift_deltas(fd, landmarks)
>>> shifts.round(3)
array([ 0.25 , -0.25 , -0.231])
```

The registered samples can be obtained with a shift

```
>>> fd.shift(shifts)
FDataGrid(...)
```

```
>>> fd = make_multimodal_samples(n_samples=3, n_modes=2, random_state=9)
>>> landmarks = make_multimodal_landmarks(n_samples=3, n_modes=2,
...                                       random_state=9)
>>> landmarks = landmarks.squeeze()
```

The function will return the registered curves

```
>>> landmark_registration(fd, landmarks)
FDataGrid(...)
```

This method will work for FDataBasis as for FDataGrids

```
>>> fd = fd.to_basis(BSpline(nbasis=12, domain_range=(-1,1)))
>>> landmark_registration(fd, landmarks)
FDataBasis(...)
```


skfda.preprocessing.registration.landmark_registration

skfda.preprocessing.registration.landmark_registration(*fd*, *landmarks*, ***, *location=None*, *eval_points=None*) [source]

Perform landmark registration of the curves.

Let t_{ij} the time where the sample i has the feature j and t_j^* the new time for the feature. The registered samples will have their features aligned, i.e., $x_i^*(t_j^*) = x_i(t_{ij})$.

See [RS05-7-3] for a detailed explanation.

Parameters:
- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List containing landmarks for each samples.
- **location** (*array_like*, *optional*) – Defines where the landmarks will be alligned. By default it will be used as location the mean of the landmarks.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain a discrete representation of the object. In case of objects with multidimensional domain a list axis with points of evaluation for each dimension.

Returns: FData with the functional data object registered.

Return type: `FData`

References:

[RS05-7-3] Ramsay, J., Silverman, B. W. (2005). Feature or landmark registration. In *Functional Data Analysis* (pp. 132-136). Springer.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_registration
>>> from skfda.representation.basis import BSpline
```

We will create a data with landmarks as example

skfda.preprocessing.registration.landmark_registration_war

skfda.preprocessing.registration.landmark_registration_warping(*fd*, *landmarks*, ***, *location=None*, *eval_points=None*) [source]

Calculate the transformation used in landmark registration.

Let t_{ij} the time where the sample i has the feature j and t_j^* the new time for the feature. The warping function will transform the new time in the old time, i.e., $h_i(t_j^*) = t_{ij}$. The registered samples can be obtained as $x_i^*(t) = x_i(h_i(t))$.

See [RS05-7-3-1] for a detailed explanation.

Parameters:
- **fd** (`FData`) – Functional data object.
- **landmarks** (*array_like*) – List containing landmarks for each samples.
- **location** (*array_like*, *optional*) – Defines where the landmarks will be alligned. By default it will be used as location the mean of the landmarks.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated to obtain a discrete representation of the object.

Returns: FDataGrid with the warpings function needed to register the functional data object.

Return type: `FDataGrid`

Raises: `ValueError` – If the object to be registered has domain dimension greater than 1 or the list of landmarks or locations does not match with the number of samples.

References:

[RS05-7-3-1] Ramsay, J., Silverman, B. W. (2005). Feature or landmark registration. In *Functional Data Analysis* (pp. 132-136). Springer.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import landmark_registration_warping
```

We will create a data with landmarks as example

```
>>> fd = make_multimodal_samples(n_samples=3, n_modes=2, random_state=9)
>>> landmarks = make_multimodal_landmarks(n_samples=3, n_modes=2,
...                                       random_state=9)
>>> landmarks = landmarks.squeeze()
```

The function will return the corresponding warping function

```
>>> warping = landmark_registration_warping(fd, landmarks)
>>> warping
FDataGrid(...)
```

The registered function can be obtained using function composition

```
>>> fd.compose(warping)
FDataGrid(...)
```

$$R^2 = \frac{\mathrm{MSE}_{phase}}{\mathrm{MSE}_{total}}$$

See [KR08-3] for a detailed explanation.

Parameters:
- **original_fdata** (`FData`) – Unregistered functions.
- **regfd** (`FData`) – Registered functions.
- **warping_function** (`FData`) – Warping functions.
- **eval_points** – (array_like, optional): Set of points where the functions are evaluated to obtain a discrete representation.

Returns: Tuple with amplitude mean square error MSE_{amp}, phase mean square error MSE_{phase}, squared correlation index R^2 and constant C_R.

Return type: `collections.namedtuple`

Raises: `ValueError` – If the curves do not have the same number of samples.

References

[KR08-3] Kneip, Alois & Ramsay, James. (2008). Quantifying amplitude and phase variation. In *Combining Registration and Fitting for Functional Models* (pp. 14-15). Journal of the American Statistical Association.

[RGS09-8-5] Ramsay J.O., Giles Hooker & Spencer Graves (2009). In *Functional Data Analysis with R and Matlab* (pp. 125-126). Springer.

Examples

```
>>> from skfda.datasets import make_multimodal_landmarks
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.preprocessing.registration import (landmark_registration_warping,
...                                mse_decomposition)
```

We will create and register data.

```
>>> fd = make_multimodal_samples(n_samples=3, random_state=1)
>>> landmarks = make_multimodal_landmarks(n_samples=3, random_state=1)
>>> landmarks = landmarks.squeeze()
>>> warping = landmark_registration_warping(fd, landmarks)
>>> fd_registered = fd.compose(warping)
>>> mse_amp, mse_pha, rsq, cr = mse_decomposition(fd, fd_registered, warping)
```

Mean square error produced by the amplitude variation.

skfda.preprocessing.registration.mse_decomposition

skfda.preprocessing.registration.mse_decomposition(*original_fdata*, *registered_fdata*, *warping_function=None*, ***, *eval_points=None*) [source]

Compute mean square error measures for amplitude and phase variation.

Once the registration has taken place, this function computes two mean squared error measures, one for amplitude variation, and the other for phase variation. It also computes a squared multiple correlation index of the amount of variation in the unregistered functions is due to phase.

Let $x_i(t), y_i(t)$ be the unregistered and registered functions respectively. The total mean square error measure (see [RGS09-8-5]) is defined as

$$\mathrm{MSE}_{total} = \frac{1}{N}\sum_{i=1}^{N}\int [x_i(t) - \bar{x}(t)]^2 dt$$

We define the constant C_R as

$$C_R = 1 + \frac{\frac{1}{N}\sum_i^N \int [Dh_i(t) - \overline{Dh}(t)][y_i^2(t) - \overline{y^2}(t)]dt}{\frac{1}{N}\sum_i^N \int y_i^2(t)dt}$$

Whose structure is related to the covariation between the deformation functions $Dh_i(t)$ and the squared registered functions $y_i^2(t)$. When these two sets of functions are independents $C_R = 1$, as in the case of shift registration.

The measures of amplitude and phase mean square error are

$$\mathrm{MSE}_{amp} = C_R \frac{1}{N}\sum_{i=1}^{N}\int \left[y_i(t) - \bar{y}(t)\right]^2 dt$$

$$\mathrm{MSE}_{phase} = \int \left[C_R \bar{y}^2(t) - \bar{x}^2(t)\right] dt$$

It can be shown that

$$\mathrm{MSE}_{total} = \mathrm{MSE}_{amp} + \mathrm{MSE}_{phase}$$

The squared multiple correlation index of the proportion of the total variation due to phase is defined as:

```
>>> f'{mse_amp:.6f}'
'0.000987'
```

In this example we can observe that the main part of the mean square error is due to the phase variation.

```
>>> f'{mse_pha:.6f}'
'0.115769'
```

Nearly 99% of the variation is due to phase.

```
>>> f'{rsq:.6f}'
'0.991549'
```

skfda.preprocessing.registration.to_srsf

skfda.preprocessing.registration.to_srsf(*fdatagrid*, *eval_points=None*) [source]

Calculate the square-root slope function (SRSF) transform.

Let $f_i : [a, b] \to \mathbb{R}$ be an absolutely continuous function, the SRSF transform is defined as

$$SRSF(f_i(t)) = sgn(f_i(t))\sqrt{|Df_i(t)|} = q_i(t)$$

This representation it is used to compute the extended non-parametric Fisher-Rao distance between functions, wich under the SRSF representation becomes the usual $\mathbb{L}^2$ distance between functions. See [SK16-4-6-1] .

Parameters:
- **fdatagrid** (`FDataGrid`) – Functions to be transformed.
- **eval_points** – (array_like, optional): Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.

Returns: SRSF functions.

Return type: `FDataGrid`

Raises: `ValueError` – If functions are multidimensional.

References

[SK16-4-6-1] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Square-Root Slope Function Representation* (pp. 91-93). Springer.

skfda.preprocessing.registration.from_srsf

skfda.preprocessing.registration.from_srsf(*fdatagrid*, *initial=None*, ***, *eval_points=None*) [source]

Given a SRSF calculate the corresponding function in the original space.

Let $f_i : [a, b] \to \mathbb{R}$ be an absolutely continuous function, the SRSF transform is defined as

$$SRSF(f_i(t)) = sgn(f_i(t))\sqrt{|Df_i(t)|} = q_i(t)$$

This transformation is a mapping up to constant. Given the srsf and the initial value the original function can be obtained as

$$f_i(t) = f(a) + \int_a^t q(t)|q(t)|dt$$

This representation it is used to compute the extended non-parametric Fisher-Rao distance between functions, wich under the SRSF representation becomes the usual $\mathbb{L}^2$ distance between functions. See [SK16-4-6-2] .

Parameters:
- **fdatagrid** (`FDataGrid`) – SRSF to be transformed.
- **initial** (*array_like*) – List of values of initial values of the original functions.
- **eval_points** – (array_like, optional): Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.

Returns: Functions in the original space.

Return type: `FDataGrid`

Raises: `ValueError` – If functions are multidimensional.

References

[SK16-4-6-2] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Square-Root Slope Function Representation* (pp. 91-93). Springer.

skfda.preprocessing.registration.elastic_registration

skfda.preprocessing.registration.elastic_registration(*fdatagrid*, *template=None*, ***, *lam=0.0*, *eval_points=None*, *fdatagrid_srsf=None*, *template_srsf=None*, *grid_dim=7*, ***kwargs*) [source]

Align a FDatagrid using the SRSF framework.

Let f be a function of the functional data object wich will be aligned to the template g. Calculates the warping wich minimises the Fisher-Rao distance between g and the registered function $f^*(t) = f(\gamma^*(t)) = f \circ \gamma^*$.

$$\gamma^* = argmin_{\gamma \in \Gamma} d_\lambda(f \circ \gamma, g)$$

Where d_λ denotes the extended Fisher-Rao distance with a penalty term, used to control the amount of warping.

$$d_\lambda^2(f \circ \gamma, g) = \|SRSF(f \circ \gamma)\sqrt{\dot{\gamma}} - SRSF(g)\|_{\mathbb{L}^2}^2 + \lambda \mathcal{R}(\gamma)$$

In the implementation it is used as penalty term

$$\mathcal{R}(\gamma) = \|\sqrt{\dot{\gamma}} - 1\|_{\mathbb{L}^2}^2$$

Wich restrict the amount of elasticity employed in the alignment.

The registered function $f^*(t)$ can be calculated using the composition $f^*(t) = f(\gamma^*(t))$.

If the template is not specified it is used the Karcher mean of the set of functions under the elastic metric to perform the alignment, wich is the local minimum of the sum of squares of elastic distances. See `elastic_mean()` .

In [SK16-4-2] are described extensively the algorithms employed and the SRSF framework.

Parameters:
- **fdatagrid** (`FDataGrid`) – Functional data object to be aligned.
- **template** (`FDataGrid` , optional) – Template to align the curves. Can contain 1 sample to align all the curves to it or the same number of samples than the fdatagrid. By default it is used the elastic mean.
- **lam** (*float*, *optional*) – Controls the amount of elasticity. Defaults to 0.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.
- **fdatagrid_srsf** (`FDataGrid` , optional) – SRSF of the fdatagrid, may be passed to avoid repeated calculation.
- **template_srsf** (`FDataGrid` , optional) – SRSF of the template, may be passed to avoid repeated calculation.
- **grid_dim** (*int*, *optional*) – Dimension of the grid used in the alignment algorithm. Defaults 7.
- ****kwargs** – Named arguments to be passed to `elastic_mean()` .

Returns: FDatagrid with the samples aligned to the template.

Return type: (`FDataGrid`)

Raises: `ValueError` – If functions are multidimensional or the number of samples are different.

References

[SK16-4-2] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Functional Data and Elastic Registration* (pp. 73-122). Springer.

skfda.preprocessing.registration.elastic_registration_warpi

skfda.preprocessing.registration.elastic_registration_warping(*fdatagrid, template=None, *, lam=0.0, eval_points=None, fdatagrid_srsf=None, template_srsf=None, grid_dim=7, **kwargs*) [source]

Calculate the warping to align a FDatagrid using the SRSF framework.

Let f be a function of the functional data object wich will be aligned to the template g. Calculates the warping wich minimises the Fisher-Rao distance between g and the registered function $f^*(t) = f(\gamma^*(t)) = f \circ \gamma^*$.

$$\gamma^* = argmin_{\gamma \in \Gamma} d_\lambda(f \circ \gamma, g)$$

Where d_λ denotes the extended amplitude distance with a penalty term, used to control the amount of warping.

$$d_\lambda^2(f \circ \gamma, g) = \|SRSF(f \circ \gamma)\sqrt{\dot{\gamma}} - SRSF(g)\|_{\mathbb{L}^2}^2 + \lambda \mathcal{R}(\gamma)$$

In the implementation it is used as penalty term

$$\mathcal{R}(\gamma) = \|\sqrt{\dot{\gamma}} - 1\|_{\mathbb{L}^2}^2$$

Wich restrict the amount of elasticity employed in the alignment.

The registered function $f^*(t)$ can be calculated using the composition $f^*(t) = f(\gamma^*(t))$.

If the template is not specified it is used the Karcher mean of the set of functions under the Fisher-Rao metric to perform the alignment, wich is the local minimum of the sum of squares of elastic distances. See `elastic_mean()`.

In [SK16-4-3] are described extensively the algorithms employed and the SRSF framework.

Parameters:
- **fdatagrid** (`FDataGrid`) – Functional data object to be aligned.
- **template** (`FDataGrid` , optional) – Template to align the curves. Can contain 1 sample to align all the curves to it or the same number of samples than the fdatagrid. By default it is used the elastic mean.
- **lam** (*float*, *optional*) – Controls the amount of elasticity. Defaults to 0.
- **eval_points** (*array_like*, *optional*) – Set of points where the functions are evaluated, by default uses the sample points of the fdatagrid.
- **fdatagrid_srsf** (`FDataGrid` , optional) – SRSF of the fdatagrid, may be passed to avoid repeated calculation.
- **template_srsf** (`FDataGrid` , optional) – SRSF of the template, may be passed to avoid repeated calculation.
- **grid_dim** (*int*, *optional*) – Dimension of the grid used in the alignment algorithm. Defaults 7.
- ****kwargs** – Named arguments to be passed to `elastic_mean()` .

Returns: Warping to align the given fdatagrid to the template.

Return type: (`FDataGrid`)

Raises: `ValueError` – If functions are multidimensional or the number of samples are different.

References

[SK16-4-3] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Functional Data and Elastic Registration* (pp. 73-122). Springer.

skfda.preprocessing.registration.warping_mean

skfda.preprocessing.registration.warping_mean(*warping, *, iter=20, tol=1e-05, step_size=1.0, eval_points=None, return_shooting=False*) [source]

Compute the karcher mean of a set of warpings.

Let $\gamma_i i = 1 ... n$ be a set of warping functions $\gamma_i : [a, b] \to [a, b]$ in Γ, i.e., monotone increasing and with the restriction $\gamma_i(a) = a\, \gamma_i(b) = b$.

The karcher mean $\bar{\gamma}$ is defined as the warping that minimises locally the sum of Fisher-Rao squared distances. [SK16-8-3-2].

$$\bar{\gamma} = argmin_{\gamma \in \Gamma} \sum_{i=1}^{n} d_{FR}^2(\gamma, \gamma_i)$$

The computation is performed using the structure of Hilbert Sphere obtained after a transformation of the warpings, see [S11-3-3].

Parameters:
- **warping** (`FDataGrid`) – Set of warpings.
- **iter** (*int*) – Maximun number of interations. Defaults to 20.
- **tol** (*float*) – Convergence criterion, if the norm of the mean of the shooting vectors, $|\bar{v}| < tol$, the algorithm will stop. Defaults to 1e-5.
- **step_size** (*float*) – Step size ϵ used to update the mean. Default to 1.
- **eval_points** (*array_like*) – Discretisation points of the warpings.
- **shooting** (*boolean*) – If true it is returned a tuple with the mean and the shooting vectors, otherwise only the mean is returned.

Returns: (`FDataGrid`) Fdatagrid with the mean of the warpings. If shooting is True the shooting vectors will be returned in a tuple with the mean.

References

[SK16-8-3-2] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Template: Center of the Mean Orbit* (pp. 274-277). Springer.

[S11-3-3] Srivastava, Anuj et. al. Registration of Functional Data Using Fisher-Rao Metric (2011). In *Center of an Orbit* (pp. 9-10). arXiv:1103.3817v2.

skfda.preprocessing.registration.elastic_mean

skfda.preprocessing.registration.elastic_mean(*fdatagrid, *, lam=0.0, center=True, iter=20, tol=0.001, initial=None, eval_points=None, fdatagrid_srsf=None, grid_dim=7, **kwargs*) [source]

Compute the karcher mean under the elastic metric.

Calculates the karcher mean of a set of functional samples in the amplitude space $\mathcal{A} = \mathcal{F}/\Gamma$.

Let q_i the corresponding SRSF of the observation f_i. The space $\mathcal{A}$ is defined using the equivalence classes $[q_i] = \{q_i \circ \gamma \| \gamma \in \Gamma\}$, where Γ denotes the space of warping functions. The karcher mean in this space is defined as

$$[\mu_q] = argmin_{[q] \in \mathcal{A}} \sum_{i=1}^{n} d_\lambda^2([q], [q_i])$$

Once $[\mu_q]$ is obtained it is selected the element of the equivalence class which makes the mean of the warpings employed be the identity.

See [SK16-8-3-1] and [S11-3].

Parameters:
- **fdatagrid** (`FDataGrid`) – Set of functions to compute the mean.
- **lam** (*float*) – Penalisation term. Defaults to 0.
- **center** (*boolean*) – If true it is computed the mean of the warpings and used to select a central mean. Defaults True.
- **iter** (*int*) – Maximun number of iterations. Defaults to 20.
- **tol** (*float*) – Convergence criterion, the algorithm will stop if :math: ´|mu^{(nu)} - mu^{(nu - 1)} |_2 / | mu^{(nu-1)} |_2 < tol´.
- **initial** (*float*) – Value of the mean at the starting point. By default takes the average of the initial points of the samples.
- **eval_points** (*array_like*) – Points of discretization of the fdatagrid.
- **fdatagrid_srsf** (`FDataGrid`) – SRSF if the fdatagrid, if it is passed it is not computed in the algorithm.
- **grid_dim** (*int*, *optional*) – Dimension of the grid used in the alignment algorithm. Defaults 7.
- **kwargs** (**) – Named options to be pased to `warping_mean()` .

Returns: FDatagrid with the mean of the functions.

Return type: (`FDataGrid`)

Raises: `ValueError` – If the object is multidimensional or the shape of the srsf do not match with the fdatagrid.

References

[SK16-8-3-1] Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In *Karcher Mean of Amplitudes* (pp. 273-274). Springer.

[S11-3] Srivastava, Anuj et. al. Registration of Functional Data Using Fisher-Rao Metric (2011). In *Karcher Mean and Function Alignment* (pp. 7-10). arXiv:1103.3817v2.

We will compute the inverse.

```
>>> inverse = invert_warping(gamma)
>>> inverse
FDataGrid(...)
```

The result of the composition should be approximately the identity function .

```
>>> identity = gamma.compose(inverse)
>>> identity([0, 0.25, 0.5, 0.75, 1]).round(3)
array([[ 0.  ,  0.25,  0.5 ,  0.75,  1.  ]])
```


skfda.preprocessing.registration.invert_warping

skfda.preprocessing.registration.invert_warping(*fdatagrid*, ***, *eval_points=None*)
[source]

Compute the inverse of a diffeomorphism.

Let $\gamma : [a, b] \to [a, b]$ be a function strictly increasing, calculates the corresponding inverse $\gamma^{-1} : [a, b] \to [a, b]$ such that $\gamma^{-1} \circ \gamma = \gamma \circ \gamma^{-1} = \gamma_{id}$.

Uses a PCHIP interpolator to compute approximately the inverse.

Parameters:
- **fdatagrid** (`FDataGrid`) – Functions to be inverted.
- **eval_points** – (array_like, optional): Set of points where the functions are interpolated to obtain the inverse, by default uses the sample points of the fdatagrid.

Returns: Inverse of the original functions.

Return type: `FDataGrid`

Raises: `ValueError` – If the functions are not strictly increasing or are multidimensional.

Examples

```
>>> import numpy as np
>>> from skfda import FDataGrid
>>> from skfda.preprocessing.registration import invert_warping
```

We will construct the warping $\gamma : [0, 1]$ wich maps t to t^3.
$rightarrow [0, 1]$

```
>>> t = np.linspace(0, 1)
>>> gamma = FDataGrid(t**3, t)
>>> gamma
FDataGrid(...)
```


skfda.preprocessing.registration.normalize_warping

skfda.preprocessing.registration.normalize_warping(*warping*, *domain_range=None*)
[source]

Rescale a warping to normalize their domain.

Given a set of warpings $\gamma_i : [a, b] \to [a, b]$ it is used an affine traslation to change the domain of the transformation to other domain, $\tilde{\gamma}_i : [\tilde{a}, \tilde{b}] \to [\tilde{a}, \tilde{b}]$.

Parameters:
- **warping** (`FDatagrid`) – Set of warpings to rescale.
- **domain_range** (*tuple*, *optional*) – New domain range of the warping. By default it is used the same domain range.

Returns: FDataGrid with the warpings normalized.

Return type: (`FDataGrid`)

skfda.ml.classification.NearestNeighbors

class **skfda.ml.classification.NearestNeighbors**(*n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Unsupervised learner for implementing neighbor searches.

Parameters:

- **n_neighbors** (*int, optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **radius** (*float, optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
- **metric** (*string or callable, (default)*) – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. Doesn't affect `fit()` method.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with 2 classes

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
```

We will fit a Nearest Neighbors estimator

```
>>> from skfda.ml.classification import NearestNeighbors
>>> neigh = NearestNeighbors(radius=.3)
>>> neigh.fit(fd)
NearestNeighbors(algorithm='auto', leaf_size=30,...)
```

Now we can query the k-nearest neighbors.

```
>>> distances, index = neigh.kneighbors(fd[:2])
>>> index # Index of k-neighbors of samples 0 and 1
array([[ 0,  7,  6, 11,  2],...)
```

```
>>> distances.round(2) # Distances to k-neighbors
array([[ 0.  ,  0.28,  0.29,  0.29,  0.3 ],
       [ 0.  ,  0.27,  0.28,  0.29,  0.3 ]])
```

We can query the neighbors in a given radius too.

```
>>> distances, index = neigh.radius_neighbors(fd[:2])
>>> index[0]
array([ 0,  2,  6,  7, 11]...)
```

```
>>> distances[0].round(2) # Distances to neighbors of the sample 0
array([ 0.  ,  0.3 ,  0.29,  0.28,  0.29])
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `RadiusNeighborsScalarRegressor`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.KNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

__init__(*n_neighbors=5, radius=1.0, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Initialize the nearest neighbors searcher.

Methods

`__init__` ([n_neighbors, radius, algorithm, ...])	Initialize the nearest neighbors searcher.
`fit` (X[, y])	Fit the model using X as training data.
`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbor
`radius_neighbors` ([X, radius, return_distance])	Finds the neighbors within a given radius of a
`radius_neighbors_graph` ([X, radius, mode])	Computes the (weighted) graph of Neighbors
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.classification.KNeighborsClassifier

class **skfda.ml.classification.KNeighborsClassifier**(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Classifier implementing the k-nearest neighbors vote.

Parameters:

- **n_neighbors** (*int, optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **weights** (*str or callable, optional (default = 'uniform')*) – weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
- **metric** (*string or callable, (default)*) – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. Doesn't affect `fit()` method.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with 2 classes

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
>>> y = 15*[0] + 15*[1]
```

We will fit a K-Nearest Neighbors classifier

```
>>> from skfda.ml.classification import KNeighborsClassifier
>>> neigh = KNeighborsClassifier()
>>> neigh.fit(fd, y)
KNeighborsClassifier(algorithm='auto', leaf_size=30,...)
```

We can predict the class of new samples

```
>>> neigh.predict(fd[::2]) # Predict labels for even samples
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
```

And the estimated probabilities.

```
>>> neigh.predict_proba(fd[0]) # Probabilities of sample 0
array([[ 1.,  0.]])
```

See also

`RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `RadiusNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.KNeighborsClassifier*.

Warning

Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor *k+1* and *k*, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

__init__(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Initialize the classifier.

Methods

`__init__` ([n_neighbors, weights, algorithm, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data and y a
`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbo
`predict` (X)	Predict the class labels for the provided data.
`predict_proba` (X)	Return probability estimates for the test data
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.classification.RadiusNeighborsClassifier

class **skfda.ml.classification.RadiusNeighborsClassifier**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_label=None, n_jobs=1, sklearn_metric=False*) [source]

Classifier implementing a vote among neighbors within a given radius

Parameters:

- **radius** (*float, optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
- **metric** (*string or callable, (default)*) – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **outlier_label** (*int, optional (default = None)*) – Label, which is given for outlier samples (samples with no neighbors on given radius). If set to None, ValueError is raised, when outlier is detected.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with 2 classes.

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
>>> y = 15*[0] + 15*[1]
```

We will fit a Radius Nearest Neighbors classifier.

```
>>> from skfda.ml.classification import RadiusNeighborsClassifier
>>> neigh = RadiusNeighborsClassifier(radius=.3)
>>> neigh.fit(fd, y)
RadiusNeighborsClassifier(algorithm='auto', leaf_size=30,...)
```

We can predict the class of new samples.

```
>>> neigh.predict(fd[::2]) # Predict labels for even samples
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
```

See also

`KNeighborsClassifier`, `KNeighborsScalarRegressor`, `RadiusNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **__init__**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_label=None, n_jobs=1, sklearn_metric=False*) [source]
>
> Initialize the classifier.

Methods

`__init__` ([radius, weights, algorithm, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data and y as
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict the class labels for the provided data.
`radius_neighbors` ([X, radius, return_distance])	Finds the neighbors within a given radius of a
`radius_neighbors_graph` ([X, radius, mode])	Computes the (weighted) graph of Neighbors
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test d
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.regression.KNeighborsScalarRegressor

class **skfda.ml.regression.KNeighborsScalarRegressor**(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Regression based on k-nearest neighbors with scalar response.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **n_neighbors** (*int, optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **weights** (*str or callable, optional (default = 'uniform')*) – weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) – Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
- **metric** (*string or callable, (default)*) – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. Doesn't affect `fit()` method.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted.

```
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.datasets import make_multimodal_landmarks
>>> y = make_multimodal_landmarks(n_samples=30, std=.5, random_state=0)
>>> y = y.flatten()
>>> fd = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
```

We will fit a K-Nearest Neighbors regressor to regress a scalar response.

```
>>> from skfda.ml.regression import KNeighborsScalarRegressor
>>> neigh = KNeighborsScalarRegressor()
>>> neigh.fit(fd, y)
KNeighborsScalarRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the modes of new samples

```
>>> neigh.predict(fd[:4]).round(2) # Predict first 4 locations
array([ 0.79,  0.27,  0.71,  0.79])
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `RadiusNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn regressor *sklearn.neighbors.KNeighborsRegressor*.

Warning

Regarding the Nearest Neighbors algorithms, if it is found that two neighbors, neighbor *k+1* and *k*, have identical distances but different labels, the results will depend on the ordering of the training data.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> ***__init__***(*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]
>
> Initialize the classifier.

Methods

`__init__` ([n_neighbors, weights, algorithm, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data and y a
`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbo
`predict` (X)	Predict the target for the provided data :para
`score` (X, y[, sample_weight])	Returns the coefficient of determination R^2
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.regression.RadiusNeighborsScalarRegressor

class **skfda.ml.regression.RadiusNeighborsScalarRegressor**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Scalar regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **radius** (*float, optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
- **metric** (*string or callable, (default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted.

```
>>> from skfda.datasets import make_multimodal_samples
>>> from skfda.datasets import make_multimodal_landmarks
>>> y = make_multimodal_landmarks(n_samples=30, std=.5, random_state=0)
>>> y = y.flatten()
>>> fd = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
```

`score` (X, y[, sample_weight])	Returns the coefficient of determination R^2 of
`set_params` (**params)	Set the parameters of this estimator.

We will fit a K-Nearest Neighbors regressor to regress a scalar response.

```
>>> from skfda.ml.regression import RadiusNeighborsScalarRegressor
>>> neigh = RadiusNeighborsScalarRegressor(radius=.2)
>>> neigh.fit(fd, y)
RadiusNeighborsScalarRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the modes of new samples.

```
>>> neigh.predict(fd[:4]).round(2) # Predict first 4 locations
array([ 0.84,  0.27,  0.66,  0.79])
```

See also

`KNeighborsClassifier`, `RadiusNeighborsClassifier`, `KNeighborsScalarRegressor`, `NearestNeighbors`, `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size`.

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> **__init__**(*radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]
>
> Initialize the classifier.

Methods

`__init__` ([radius, weights, algorithm, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data and y as t
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict the target for the provided data :param
`radius_neighbors` ([X, radius, return_distance])	Finds the neighbors within a given radius of a fc
`radius_neighbors_graph` ([X, radius, mode])	Computes the (weighted) graph of Neighbors fc

skfda.ml.regression.RadiusNeighborsFunctionalRegresso

class **skfda.ml.regression.RadiusNeighborsFunctionalRegressor**(*radius=1.0, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_response=None, n_jobs=1, sklearn_metric=False*) [source]

Functional regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **radius** (*float, optional (default = 1.0)*) – Range of parameter space to use by default for `radius_neighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **regressor** (*callable, optional ((default =)* – `mean`)) Function to perform the local regression. By default used the mean. Can accept a user-defined function wich accepts a `FDataGrid` with the neighbors of a test sample, and if weights != 'uniform' an array of weights as second parameter.
- **algorithm** ({*'auto', 'ball_tree', 'brute'*}, *optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree`.
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
- **metric** (*string or callable, (default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **outlier_response** (`FDataGrid` , optional (default = None)) – Default response for test samples without neighbors.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics`.

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted, and we will try to predict 5 X +1.

```
>>> from skfda.datasets import make_multimodal_samples
>>> X_train = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
>>> y_train = 5 * X_train + 1
>>> X_test = make_multimodal_samples(n_samples=5, std=.5, random_state=0)
```

We will fit a Radius Nearest Neighbors functional regressor.

```
>>> from skfda.ml.regression import RadiusNeighborsFunctionalRegressor
>>> neigh = RadiusNeighborsFunctionalRegressor(radius=.03)
>>> neigh.fit(X_train, y_train)
RadiusNeighborsFunctionalRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the response of new samples.

```
>>> neigh.predict(X_test)
FDataGrid(...)
```

See also

`KNeighborsClassifier` , `RadiusNeighborsClassifier` , `KNeighborsScalarRegressor` , `NearestNeighbors` , `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size` .

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

> ***__init__***(*radius=1.0, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, outlier_response=None, n_jobs=1, sklearn_metric=False*) [source]
>
> Initialize the classifier.

Methods

`__init__` ([radius, weights, regressor, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data.
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict functional responses.
`radius_neighbors` ([X, radius, return_distance])	Finds the neighbors within a given radius of a fdatag
`radius_neighbors_graph` ([X, radius, mode])	Computes the (weighted) graph of Neighbors for poi
`score` (X, y)	TODO
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.regression.KNeighborsFunctionalRegressor

class **skfda.ml.regression.KNeighborsFunctionalRegressor**(*n_neighbors=5, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*) [source]

Functional regression based on neighbors within a fixed radius.

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Parameters:

- **n_neighbors** (*int, optional (default = 5)*) – Number of neighbors to use by default for `kneighbors()` queries.
- **weights** (*str or callable*) –
  weight function used in prediction. Possible values:
  - 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
  - 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
  - [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

  Uniform weights are used by default.
- **regressor** (*callable, optional ((default =)* – `mean`)) Function to perform the local regression. By default used the mean. Can accept a user-defined function wich accepts a `FDataGrid` with the neighbors of a test sample, and if weights != 'uniform' an array of weights as second parameter.
- **algorithm** (*{'auto', 'ball_tree', 'brute'}, optional*) –
  Algorithm used to compute the nearest neighbors:
  - 'ball_tree' will use `sklearn.neighbors.BallTree` .
  - 'brute' will use a brute-force search.
  - 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit()` method.
- **leaf_size** (*int, optional (default = 30)*) – Leaf size passed to BallTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.
- **metric** (*string or callable, (default)* – `lp_distance`) the distance metric to use for the tree. The default metric is the Lp distance. See the documentation of the metrics module for a list of available metrics.
- **metric_params** (*dict, optional (default = None)*) – Additional keyword arguments for the metric function.
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.
- **sklearn_metric** (*boolean, optional (default = False)*) – Indicates if the metric used is a sklearn distance between vectors (see `sklearn.neighbors.DistanceMetric`) or a functional metric of the module `skfda.misc.metrics` .

Examples

Firstly, we will create a toy dataset with gaussian-like samples shifted, and we will try to predict 5 X +1.

```
>>> from skfda.datasets import make_multimodal_samples
>>> X_train = make_multimodal_samples(n_samples=30, std=.5, random_state=0)
>>> y_train = 5 * X_train + 1
>>> X_test = make_multimodal_samples(n_samples=5, std=.5, random_state=0)
```

We will fit a K-Nearest Neighbors functional regressor.

```
>>> from skfda.ml.regression import KNeighborsFunctionalRegressor
>>> neigh = KNeighborsFunctionalRegressor()
>>> neigh.fit(X_train, y_train)
KNeighborsFunctionalRegressor(algorithm='auto', leaf_size=30,...)
```

We can predict the response of new samples.

```
>>> neigh.predict(X_test)
FDataGrid(...)
```

See also

`KNeighborsClassifier` , `RadiusNeighborsClassifier` , `KNeighborsScalarRegressor` , `NearestNeighbors` , `NearestCentroids`

Notes

See Nearest Neighbors in the sklearn online documentation for a discussion of the choice of `algorithm` and `leaf_size` .

This class wraps the sklearn classifier *sklearn.neighbors.RadiusNeighborsClassifier*.

https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

***__init__*(*n_neighbors=5, weights='uniform', regressor=<function mean>, algorithm='auto', leaf_size=30, metric=<function lp_distance>, metric_params=None, n_jobs=1, sklearn_metric=False*)**
[source]

Initialize the classifier.

Methods

`__init__` ([n_neighbors, weights, regressor, ...])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data.
`get_params` ([deep])	Get parameters for this estimator.
`kneighbors` ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
`kneighbors_graph` ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbo
`predict` (X)	Predict functional responses.
`score` (X, y)	TODO
`set_params` (**params)	Set the parameters of this estimator.

skfda.ml.classification.NearestCentroids

class **skfda.ml.classification.NearestCentroids**(*metric=<function lp_distance>, mean=<function mean>*) [source]

Nearest centroid classifier for functional data.

Each class is represented by its centroid, with test samples classified to the class with the nearest centroid.

Parameters:

- **metric** (*callable, (default)* – `lp_distance`) The metric to use when calculating distance between test samples and centroids. See the documentation of the metrics module for a list of available metrics. Defaults used L2 distance.
- **mean** (callable, (default `mean`)) – The centroids for the samples corresponding to each class is the point from which the sum of the distances (according to the metric) of all samples that belong to that particular class are minimized. By default it is used the usual mean, which minimizes the sum of L2 distance. This parameter allows change the centroid constructor. The function must accept a `FData` with the samples of one class and return a `FData` object with only one sample representing the centroid.

centroids_

`FDataGrid` – FDatagrid containing the centroid of each class

Examples

Firstly, we will create a toy dataset with 2 classes

```
>>> from skfda.datasets import make_sinusoidal_process
>>> fd1 = make_sinusoidal_process(phase_std=.25, random_state=0)
>>> fd2 = make_sinusoidal_process(phase_mean=1.8, error_std=0.,
...                               phase_std=.25, random_state=0)
>>> fd = fd1.concatenate(fd2)
>>> y = 15*[0] + 15*[1]
```

We will fit a Nearest centroids classifier

```
>>> from skfda.ml.classification import NearestCentroids
>>> neigh = NearestCentroids()
>>> neigh.fit(fd, y)
NearestCentroids(...)
```

We can predict the class of new samples

```
>>> neigh.predict(fd[::2]) # Predict labels for even samples
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1])
```

See also

`KNeighborsClassifier` , `RadiusNeighborsClassifier` , `KNeighborsScalarRegressor` , `RadiusNeighborsScalarRegressor` , `NearestNeighbors`

__init__(*metric=<function lp_distance>, mean=<function mean>*) [source]

Initialize the classifier.

Methods

`__init__` ([metric, mean])	Initialize the classifier.
`fit` (X, y)	Fit the model using X as training data and y as target values.
`get_params` ([deep])	Get parameters for this estimator.
`predict` (X)	Predict the class labels for the provided data.
`score` (X, y[, sample_weight])	Returns the mean accuracy on the given test data and labels.
`set_params` (**params)	Set the parameters of this estimator.

skfda.datasets.make_sinusoidal_process

skfda.datasets.make_sinusoidal_process(*n_samples: int = 15, n_features: int = 100, *, start: float = 0.0, stop: float = 1.0, period: float = 1.0, phase_mean: float = 0.0, phase_std: float = 0.6, amplitude_mean: float = 1.0, amplitude_std: float = 0.05, error_std: float = 0.2, random_state=None*) [source]

Generate sinusoidal proccess.

Each sample $x_i(t)$ is generated as:

$$x_i(t) = \alpha_i \sin(\omega t + \phi_i) + \epsilon_i(t)$$

where $\omega = \frac{2\pi}{\text{period}}$. Amplitudes α_i and phases ϕ_i are normally distributed. $\epsilon_i(t)$ is a gaussian white noise process.

Parameters:
- **n_samples** – Total number of samples.
- **n_features** – Points per sample.
- **start** – Starting point of the samples.
- **stop** – Ending point of the samples.
- **period** – Period of the sine function.
- **phase_mean** – Mean of the phase.
- **phase_std** – Standard deviation of the phase.
- **amplitude_mean** – Mean of the amplitude.
- **amplitude_std** – Standard deviation of the amplitude.
- **error_std** – Standard deviation of the gaussian Noise.
- **random_state** – Random state.

Returns: `FDataGrid` object comprising all the samples.

skfda.datasets.make_multimodal_samples

skfda.datasets.make_multimodal_samples(*n_samples: int = 15, *, n_modes: int = 1, points_per_dim: int = 100, ndim_domain: int = 1, ndim_image: int = 1, start: float = -1, stop: float = 1.0, std: float = 0.05, mode_std: float = 0.02, noise: float = 0.0, modes_location=None, random_state=None*) [source]

Generate multimodal samples.

Each sample $x_i(t)$ is proportional to a gaussian mixture, generated as the sum of multiple pdf of multivariate normal distributions with different means.

$$x_i(t) \propto \sum_{n=1}^{\text{n\_modes}} \exp\left(-\frac{1}{2\sigma}(t-\mu_n)^T \mathbb{1}(t-\mu_n)\right)$$

Where $\mu_n = \text{mode\_location}_n + \epsilon$ and ϵ is normally distributed, with mean $\mathbb{0}$ and standard deviation given by the parameter *std*.

Parameters:
- **n_samples** – Total number of samples.
- **n_modes** – Number of modes of each sample.
- **points_per_dim** – Points per sample. If the object is multidimensional indicates the number of points for each dimension in the domain. The sample will have :math: *text{points_per_dim}^text{ndim_domain}* points of discretization.
- **ndim_domain** – Number of dimensions of the domain.
- **ndim_image** – Number of dimensions of the image
- **start** – Starting point of the samples. In multidimensional objects the starting point of each axis.
- **stop** – Ending point of the samples. In multidimensional objects the ending point of each axis.
- **std** – Standard deviation of the variation of the modes location.
- **mode_std** – Standard deviation σ of each mode.
- **noise** – Standard deviation of Gaussian noise added to the data.
- **modes_location** – List of coordinates of each mode.
- **random_state** – Random state.

Returns: `FDataGrid` object comprising all the samples.

skfda.datasets.make_multimodal_landmarks

skfda.datasets.make_multimodal_landmarks(*n_samples: int = 15, *, n_modes: int = 1, ndim_domain: int = 1, ndim_image: int = 1, start: float = -1, stop: float = 1, std: float = 0.05, random_state=None*) [source]

Generate landmarks points.

Used by `make_multimodal_samples()` to generate the location of the landmarks.

Generates a matrix containing the landmarks or locations of the modes of the samples generates by `make_multimodal_samples()` .

If the same random state is used when generating the landmarks and multimodal samples, these will correspond to the position of the modes of the multimodal samples.

Parameters:
- **n_samples** – Total number of samples.
- **n_modes** – Number of modes of each sample.
- **ndim_domain** – Number of dimensions of the domain.
- **ndim_image** – Number of dimensions of the image
- **start** – Starting point of the samples. In multidimensional objects the starting point of the axis.
- **stop** – Ending point of the samples. In multidimensional objects the ending point of the axis.
- **std** – Standard deviation of the variation of the modes location.
- **random_state** – Random state.

Returns: `np.ndarray` with the location of the modes, where the component (i,j,k) corresponds to the mode k of the image dimension j of the sample i.

skfda.datasets.make_random_warping

skfda.datasets.make_random_warping(*n_samples: int = 15, n_features: int = 100, *, start: float = 0.0, stop: float = 1.0, sigma: float = 1.0, shape_parameter: float = 50, n_random: int = 4, random_state=None*) [source]

Generate random warping functions.

Let $v(t)$ be a randomly generated function defined in $[0, 1]$

$$v(t) = \sum_{j=0}^{N} a_j \sin(\frac{2\pi j}{K}t) + b_j \cos(\frac{2\pi j}{K}t)$$

where $a_j, b_j \sim N(0, \sigma)$.

The random warping it is constructed making an exponential map to Γ.

$$\gamma(t) = \int_0^t \left(\frac{\sin(\|v\|)}{\|v\|}v(s) + \cos(\|v\|)\right)^2 ds$$

An affine traslation it is used to define the warping in $[a, b]$.

The smoothing and shape of the warpings can be controlling changing N, σ and $K = 1 + \text{shape\_parameter}$.

Parameters:

- **n_samples** – Total number of samples. Defaults 15.
- **n_features** – The total number of trajectories. Defaults 100.
- **start** – Starting point of the samples. Defaults 1.
- **stop** – Ending point of the samples. Defaults 0.
- **sigma** – Parameter to control the variance of the samples. Defaults 1.
- **shape_parameter** – Parameter to control the shape of the warpings. Should be a positive value. When the shape parameter goes to infinity the warpings generated are γ_{id}. Defaults to 50.
- **n_random** – Number of random sines and cosines to be sum to construct the warpings.
- **random_state** – Random state.

Returns: `FDataGrid` object comprising all the samples.