

Degree work

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



Pablo Marcos Manchón

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



Double degree in Computer Engineering and Mathematics

DEGREE WORK

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

Pablo Marcos Manchón

PRINTED IN SPAIN

*Lo peor es cuando has terminado un capítulo
y la máquina de escribir no aplaude.*

Orson Welles

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 Luís, por sus inestimables aportaciones matemáticas. Finalmente a Pablo 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 de mi vida. 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, también denominado FDA, es una rama de la estadística encargada del estudio de variables aleatorias de naturaleza funcional, como puede ser un conjunto de series temporales o de curvas en el espacio.

En las últimas dos décadas las técnicas empleadas para este análisis han evolucionado rápidamente, así como sus aplicaciones en gran cantidad de campos como medicina, bioinformática o ingeniería.

Sin embargo, debido a la relativamente reciente aparición de este campo, no hay una gran diversidad de soluciones software que engloben estas técnicas. Por esta razón en el año 2017 surgió el proyecto de código abierto llamado *scikit-fda*, con el objetivo de crear un paquete en Python que diera soporte a este estudio, junto con una comunidad que contribuyera a su desarrollo.

Con este trabajo se pretende contribuir al crecimiento de este proyecto, cuyo objetivo a largo plazo es convertirse en una referencia en FDA que contenga una amplia variedad de herramientas para el estudio de estos datos.

En concreto, durante este trabajo se han realizado contribuciones en diferentes áreas de este campo, entre las que se encuentran el registro de los datos, etapa en la cual se estudia la variabilidad debida a su escala interna, como puede ser el tiempo en el que evoluciona un proceso; o su representación, en la cual se estudian los problemas asociados su tratamiento y almacenamiento.

PALABRAS CLAVE

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

ABSTRACT

Functional data analysis, also called FDA, is a branch of statistics that deals with the study of random variables of a functional nature, such as temporal series or curves in space.

In the last two decades, techniques used for this analysis have evolved quickly, as well as their applications in many fields such as medicine, bioinformatics or engineering.

However, due to the relatively recent appearance of this field, there is not a great variety of software solutions that encompass these techniques. For this reason, in 2017 the open-source project called *scikit-fda* arose, with the aim of creating a Python package which would support this study, along with a community that would contribute to its development.

This work aims to contribute to the growth of this project, whose long-term goal is to become a reference in FDA, containing a wide variety of tools for the study of these data.

Specifically, during this work contributions have been made in different areas of this field, including the registration of data, a stage in which the variability due to its internal scale is studied, such as the time in which a process evolves; or its representation, in which the problems associated with its treatment and storage are studied.

KEYWORDS

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

TABLE OF CONTENTS

1	Introduction	1
1.1	Goals and scope	2
1.2	Document Structure	2
2	State of the art: registration, interpolation and nearest neighbors	3
2.1	Functional data representation	3
2.1.1	Interpolation	4
2.2	Registration	7
2.2.1	Shift registration	8
2.2.2	Warping functions	9
2.2.3	Landmark registration	10
2.2.4	Pairwise and groupwise alignment	11
2.2.5	Amplitude phase decomposition	13
2.3	Elastic methods in functional data analysis	14
2.3.1	Fisher-Rao metric	14
2.3.2	Amplitude and phase spaces	16
2.3.3	Pairwise alignment	19
2.3.4	Karcher means	19
2.3.5	Elastic registration	21
2.3.6	Restricting elasticity	22
2.4	Nearest neighbors	23
2.4.1	Nearest neighbor search	23
2.4.2	Classification	24
2.4.3	Regression	24
3	Design and development	25
3.1	Analysis	25
3.2	Design	26
3.2.1	Representation module	26
3.2.2	Preprocessing module	27
3.2.3	Machine learning module	27
3.2.4	Miscellaneous module	27
3.2.5	Dataset module	27
3.3	Coding, documenting and testing	27

3.4 Development, version control and continuous integration	29
4 Conclusions and future work	31
Bibliography	34
Appendices	35
A Algorithms and proofs	37
A.1 Shift registration by the Newton-Raphson algorithm	37
A.2 Dynamic programming algorithm	38
A.3 Karcher means computation	38
A.4 Proofs of some mathematical results	38
B Example notebooks	41
C Programmer's guide	67

LISTS

List of equations

2.1	Linear interpolation	4
2.2	Shift registration	8
2.3	Least square criterion	8
2.4	Warping registration	9
2.5	Warping mapping of landmarks	10
2.6	Lack of symmetry	12
2.7	Mean square error total	13
2.8	Mean square decomposed	13
2.9	Square multiple correlation index	13
2.10	Fisher-Rao metric	14
2.11	Length of path	15
2.12	Length of shortest path	15
2.13	Straight line of SRSFs	15
2.14	Elastic distance	18
2.15	Norm of warping	18
2.16	Phase distance	18
2.17	Relative phase	18
2.18	Pairwise alignment with F-R metric	19
2.19	Inverse consistency	19
2.20	Karcher means	19
2.21	Karcher mean on \mathcal{A}	21
2.22	Identity mean	21
2.23	Restricted amplitude distance	22
2.24	Classification prediction	24
2.25	Regression response	24
A.1	First derivative of REGSSE	37
A.2	Second derivative of REGSSE	37
A.3	Approximation of second derivative of REGSSE	37

List of figures

2.1	Function resampled using interpolation	4
2.2	Example of linear interpolation	5
2.3	Example of spline interpolation	5
2.4	Interpolation of surface	6
2.5	Example of smoothing	7
2.6	Male growth rate	7
2.7	Amplitude and phase variability	8
2.8	Shift registration of a dataset	9
2.9	Set of warping functions	10
2.10	Shift registration of a dataset	11
2.11	Pairwise alignment	11
2.12	Pinching force effect	12
2.13	Geodesic path in \mathcal{F}	15
2.14	Action of Γ	16
2.15	Rotation in complex plane	16
2.16	Phase and amplitude in the complex plane	17
2.17	Functions in the same orbit	17
2.18	Karcher mean of dataset	20
2.19	Scheme of the elastic registration procedure	21
2.20	Elastic registration of the Berkeley velocity curves	22
2.21	Penalized elastic registration	22
2.22	Neighborhoods using distance \mathbb{L}^∞	23
3.1	scikit-fda logo	25
3.2	Map of scikit-fda	26
3.3	Scikit-fda online documentation	28
3.4	Example of git flow branches	29
3.5	Example of travis checks	30

INTRODUCTION

Functional data analysis (FDA), is a branch of the statistic which 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 so 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.

Although there were some software solutions to support this field, such as *fda* [2], powered by J.O. Ramsay [3]; or *fda.usc* [4], powered by the University of Santiago de Compostela. But none of them had been implemented under the philosophy of a open-source project, encouraging open collaboration. In addition all implementations were based on R or Matlab, and there were no Python packages for this purpose, although its use has become widespread in recent years in statistics and machine learning.

In 2017, the *scikit-fda* project came into being, under the name *fda* [5], as part of a degree work 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 a open-source project, along with the creation of a community that contributes to development and maintenance. Currently the project is being driven by the machine learning group of the UAM (GAA-UAM) together with the contributions of several degree works, including this one.

Although the project is in an early stage of development, 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 continue the *scikit-fda* project, collaborating with in a general way in its development, but especially in the registration of functional data, one of the areas of FDA, in which it is studied the variability of the data due to its internal scale, such as time in which a process unfolds.

Due to the extent of this area, it is not possible to cover all techniques in this field, therefore the purpose of this work was not delimited from the beginning in a clear way. It was decided to start with techniques exposed in the book *Functional data analysis* [7], which makes a general overview of different aspects in FDA, but without a clear goal on the specific aspects to be dealt with.

For this reason, weekly meetings were scheduled throughout the year with all the team involved in the project, to review the scope of this work and the next steps to follow in its development, with the goal in mind to contribute as much as possible to the growth of this project.

1.2. Document Structure

The main part of the present document is divided into four 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.

STATE OF THE ART: REGISTRATION, INTERPOLATION AND NEAREST NEIGHBORS

In FDA, the analyzed data might take the form of temporal series, curves, surfaces or anything else varying over a continuum. The observations that will make up our data will come from random variables, whose realizations will take values in functional spaces, that is to say, they will be realizations of stochastic processes. For this reason, under this framework, we will consider them functions.

One may wonder what makes FDA different from multivariate analysis, considering the fact that data is generally collected and stored in the form of discrete point sets (t_j, y_j) [8]. In multivariate statistics one works with the vector y_j , applying statistical methods and vector calculus to its study. 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.

In this chapter we will make an overview, from a theoretical point of view, of different FDA topics covered in this work.

2.1. Functional data representation

Since our observations will be functions, dependent on one or several continuous parameters, such as the time in which a process takes place, it will be impossible to measure and store all the points which they are defined for.

In practice, the information of a functional datum $x(t)$ is observed and recorded as pairs $x(t_j) = (t_j, y_j)$, where y_j is a snapshot of the function x at t_j , possibly blurred by measurement error [7].

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

There are two main approaches to represent the data. The first, called discrete representation, stores a finite grid of pairs (t_j, y_j) with the recorded values, but unlike multivariate statistics, the continuous dependence between y_j and t_j will be taken into account.

The second one is a parametric approach, called basis representation, in which we will define a

system of functions $\{\Phi_n(t)\}_{n=1}^N$, such as a truncated Fourier basis or a set of polynomials. This system will form a finite subspace of the original functional space, where we will project the observations, obtaining a representation associated to the coefficients of the basis $x(t) \approx \sum_{n=1}^N c_n \Phi_n(t)$.

2.1.1. Interpolation

In the discrete representation, frequently, it will be necessary to resample or evaluate the data, or its derivatives, at points within the domain range, different than the original pairs $\{(t_j, y_j)\}$ at which our observations have been measured. An example of this is shown in the figure 2.1. For this purpose, we will use interpolation.

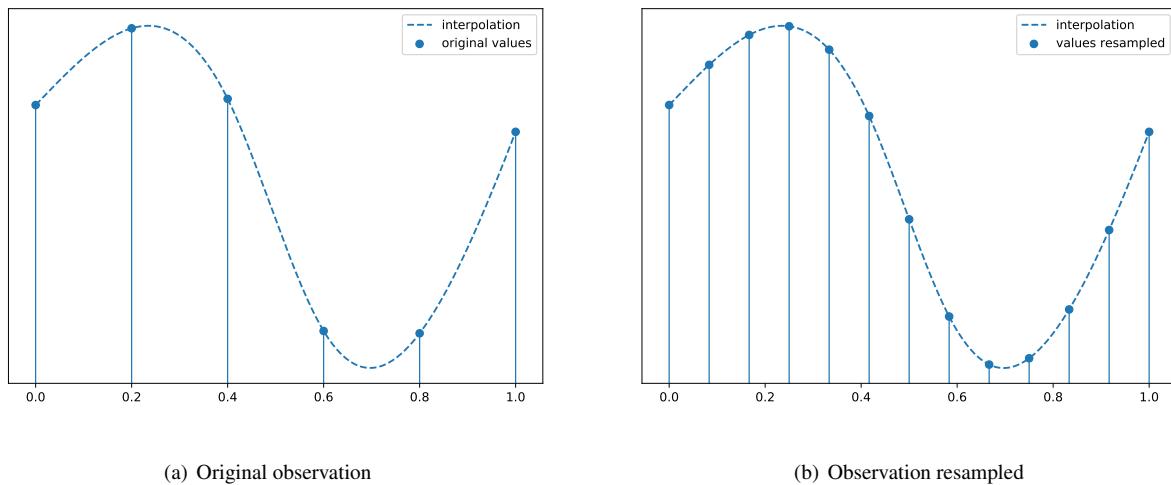


Figure 2.1: Function resampled using interpolation

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.

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 type of interpolation, we will obtain a piecewise linear function, whose values in the interval $[t_k, t_{k+1}]$ will be given by

$$x(t) = y_k + \frac{y_{k+1} - y_k}{t_{k+1} - t_k} (t - t_k). \quad (2.1)$$

In the case of multivariate functions, multilinear interpolation generalizes this method to higher dimensions. The same principles will be applied, but the piecewise domain will be composed by regions calculated using triangulation, and the evaluation will be 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.

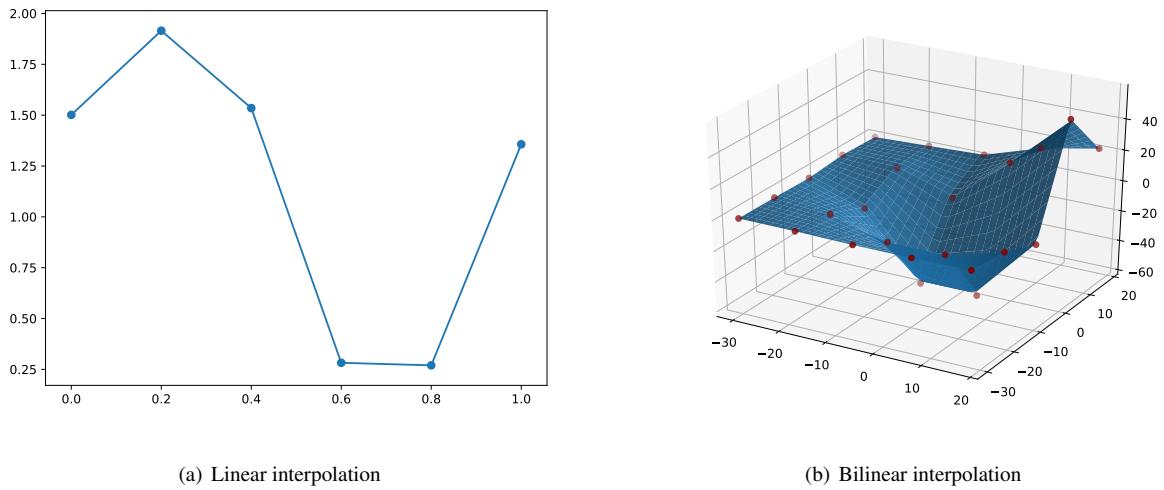


Figure 2.2: Example of linear interpolation

Spline interpolation

In spline interpolation [9], as in the linear case, we will define a piecewise function, but employing polynomials to join the different points known. The main advantage of using splines of order n , i.e., using polynomial of order n in the different intervals, is that we will not only obtain continuous functions, as in the linear case, but we will also obtain functions $C^{n-1}[a, b]$. This fact will be crucial in FDA, because we will need to use derivatives in many phases of the analysis.

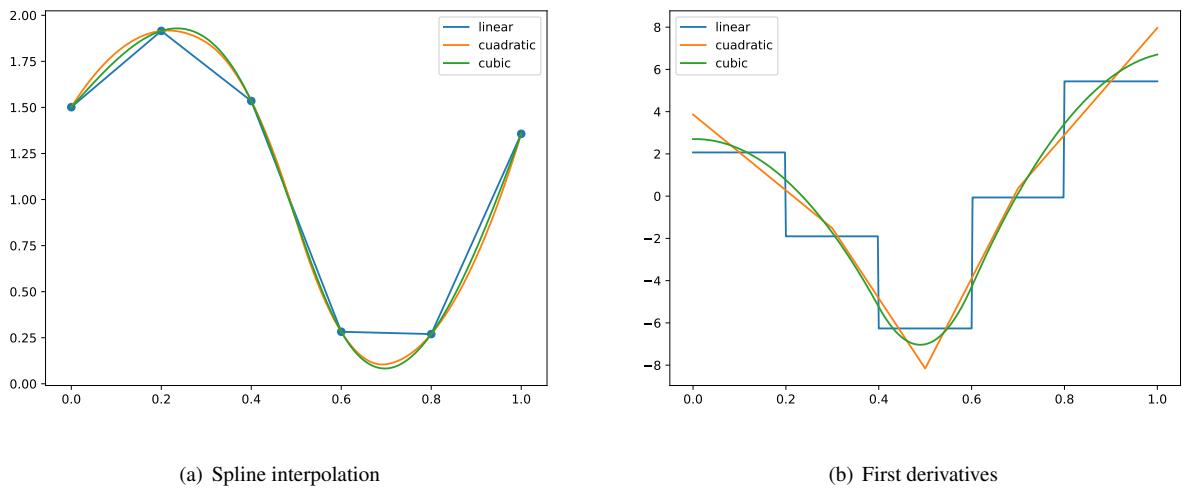


Figure 2.3: Example of spline interpolation

To achieve this, we will have to match the values of the derivatives of the adjacent splines in the interpolation knots. If we denote by $p_k(t) = \sum_{j=0}^n c_{jk} t^k$ 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 $p_k^{(d)}(t_k) = p_{k+1}^{(d)}$ for $d = 1, \dots, n - 1$. For this purpose, we will define a linear system of equations which will 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 $n - 1$ due to the derivation of the polynomials.

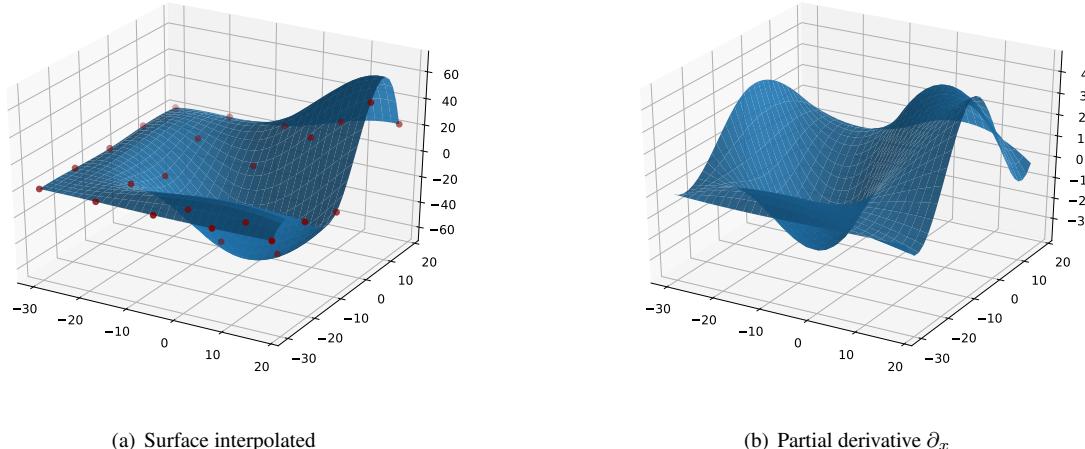


Figure 2.4: Interpolation of surface with bicubic splines

We can extend spline interpolation for multivariate functions, such as surfaces, where bivariate splines will be used. In this case, it is used triangulation to calculate the regions in which the polynomials will be defined, which will be of the form $p(x, y) = \sum_{0 \leq i+j \leq n} c_{i,j} x^i y^j$. 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 x , of the surface.

Smoothing spline interpolation

In several contexts, the functional data will present noise, due to the measurement procedure or to the intrinsic random behaviour of the process, this will motivate the use of smoothing splines.

This method is a variation of the spline interpolation, but we will not use directly the points $\{t_k\}_{k=1}^N$ as knots. Instead, we will use a smaller set of knots $\{\tilde{t}_k\}_{k=1}^M$, generally different from the originals, with their corresponding values $\tilde{y}_k = x(\tilde{t}_k)$ calculated using spline interpolation. The function interpolated using smoothing splines will be 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.

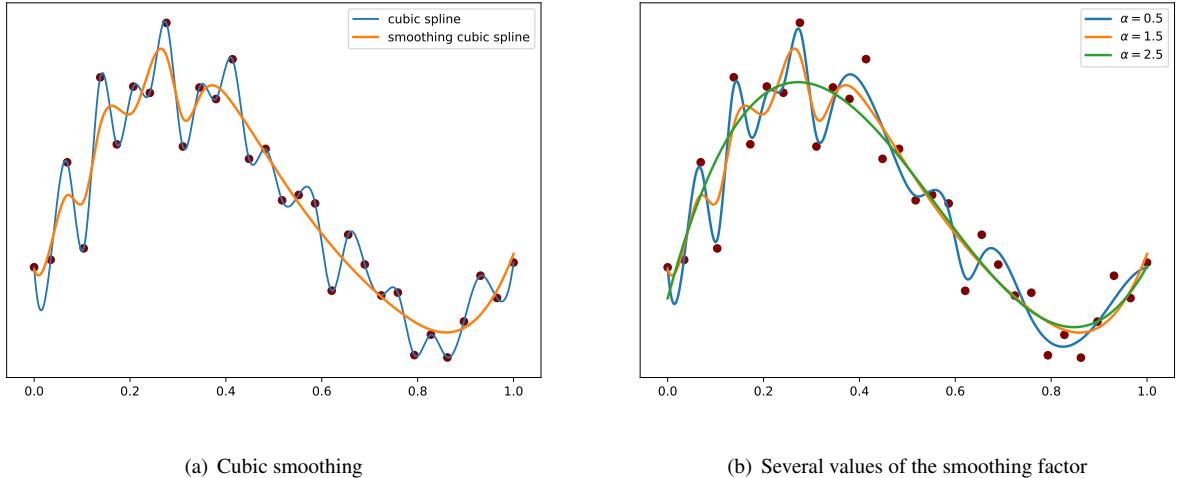


Figure 2.5: Smoothing interpolation

2.2. Registration

Once our data is in functional form, we may encounter an interesting phenomenon that creates problems in FDA. In many situations, functional observations have similar shapes, but they are in some way misaligned. This variability will interfere with further analysis, and therefore, a registration of our data will be necessary in order to eliminate and quantify its variation.

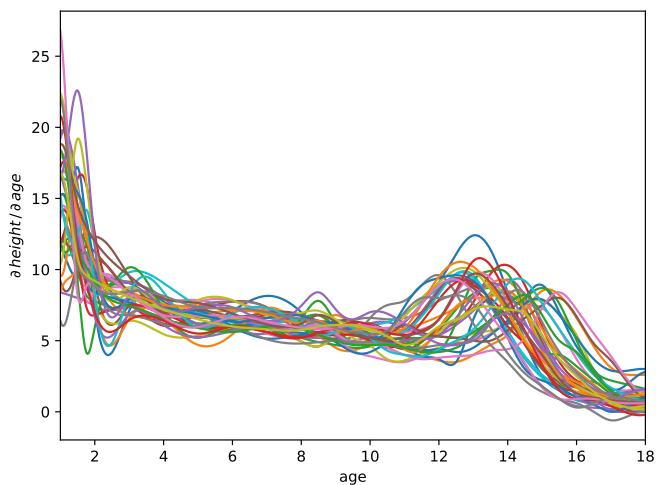


Figure 2.6: Male growth rate

The Berkeley Growth Study [10], performed in 1954, recorded the heights of 54 girls and 39 boys between the ages of 1 and 18 years. In the figure 2.6 it is shown the velocity growth curves of the boys. 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, so 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 decomposed into two sources; the first one comes from the random curve-to-curve variation [11], this is called **amplitude variation**. In the figure 2.21(a) it is shown a set of curves whose variability proceeds exclusively from the amplitude. The second type proceeds from misalignments of the curves with respect to the domain; this is called **phase variation**. In the figure 2.21(b) it is shown a set of curves whose variation is completely due to the phase; this source of variability is the one that will be dealt with in the registration process.

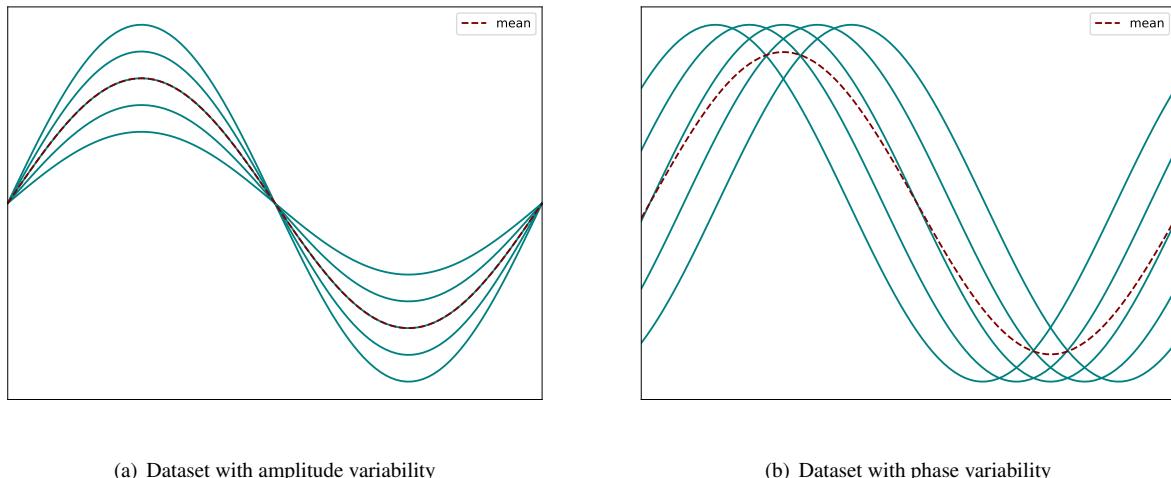


Figure 2.7: Amplitude and phase variability

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

$$x_i^*(t) = x_i(t + \delta_i) \quad i = 1, 2, \dots, n \quad (2.2)$$

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, defined as

$$\text{REGSSE} = \sum_{i=1}^n \int_{\mathcal{T}} [x_i(t + \delta_i) - \hat{\mu}(t)]^2 dt \quad (2.3)$$

The alignment problem will be based on finding the values δ_i that minimize \mathbb{L}^2 distance to the cross-sectional mean. For this purpose, we will use the derivatives of the functions x_i , using a variation of the Newton-Raphson method. A description of the algorithm used to calculate these values is given in the Appendix A.1.

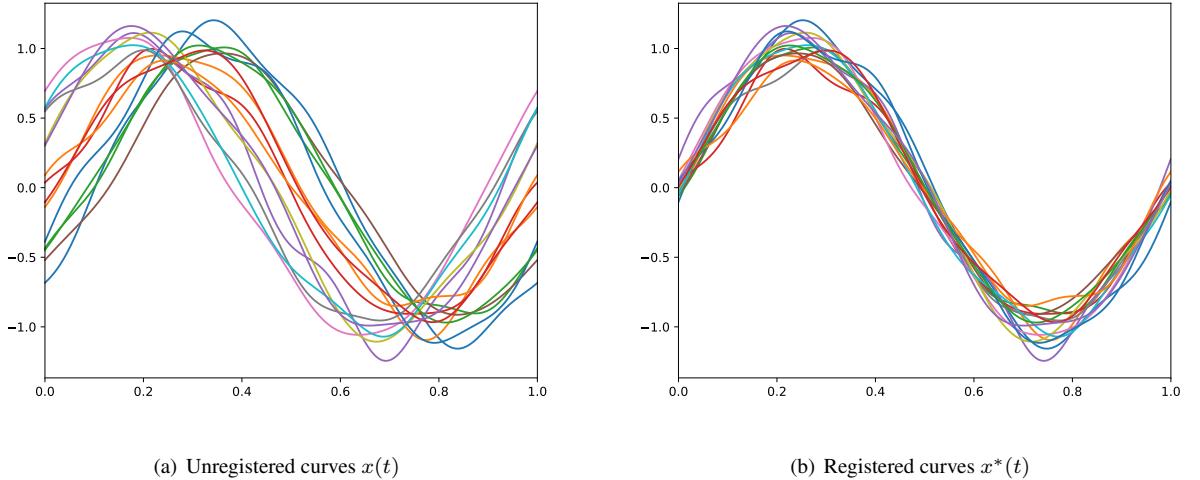


Figure 2.8: Shift registration of a dataset

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 data, we will have functions $x_i : \mathcal{T} \rightarrow \mathbb{R}$, 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 $\gamma_i : \mathcal{T} \rightarrow \mathcal{T}$, referred to as warping functions in the related literature, that we will use to reparametrize the domain, through which we will be able to obtain the curves registered by means of composition of functions, i.e.,

$$x_i^*(t) = x_i(\gamma_i(t)) = x_i \circ \gamma_i. \quad (2.4)$$

So that the alignment does not alter the structure of our functional data, these functions γ_i must be boundary-preserving diffeomorphisms. 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.

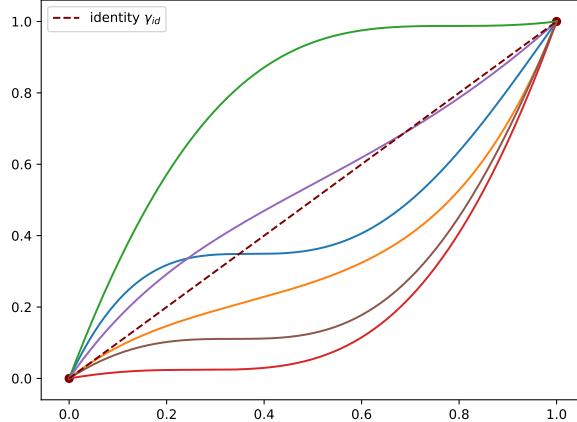


Figure 2.9: Set of warping functions defined in $\mathcal{T} = [0, 1]$

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 these warping functions as Γ .

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, 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.

The landmark registration process will require, for each observation x_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

$$x_i^*(t_j^*) = x_i(\gamma_i(t_j^*)) = x_i(t_{ij}). \quad (2.5)$$

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.

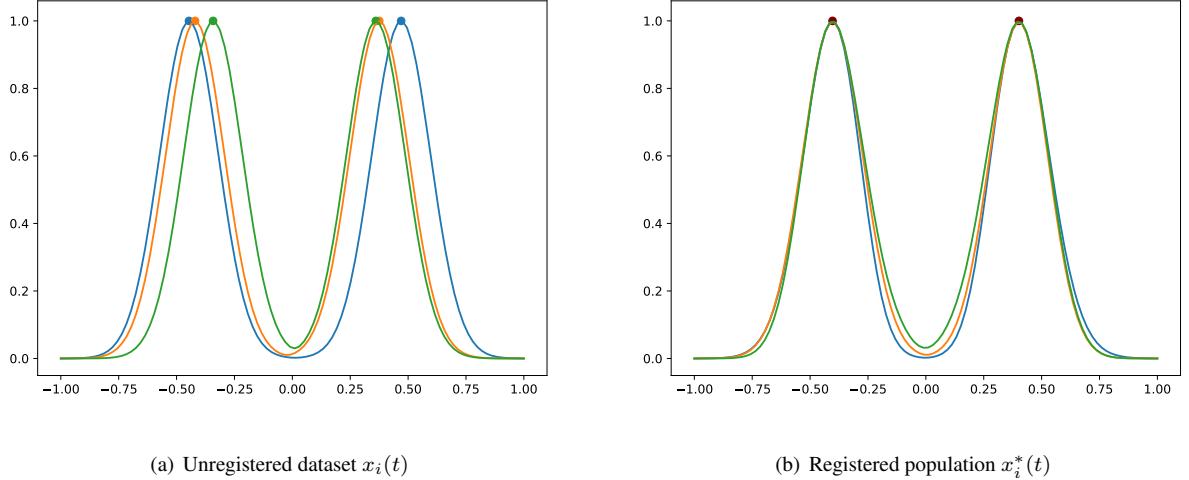


Figure 2.10: Shift registration of a dataset

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 $E : \mathcal{F} \times \mathcal{F} \rightarrow \mathbb{R}^+$ 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., $\gamma^* = \operatorname{argmin}_{\gamma \in \Gamma} E[f_1, f_2 \circ \gamma]$. 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.

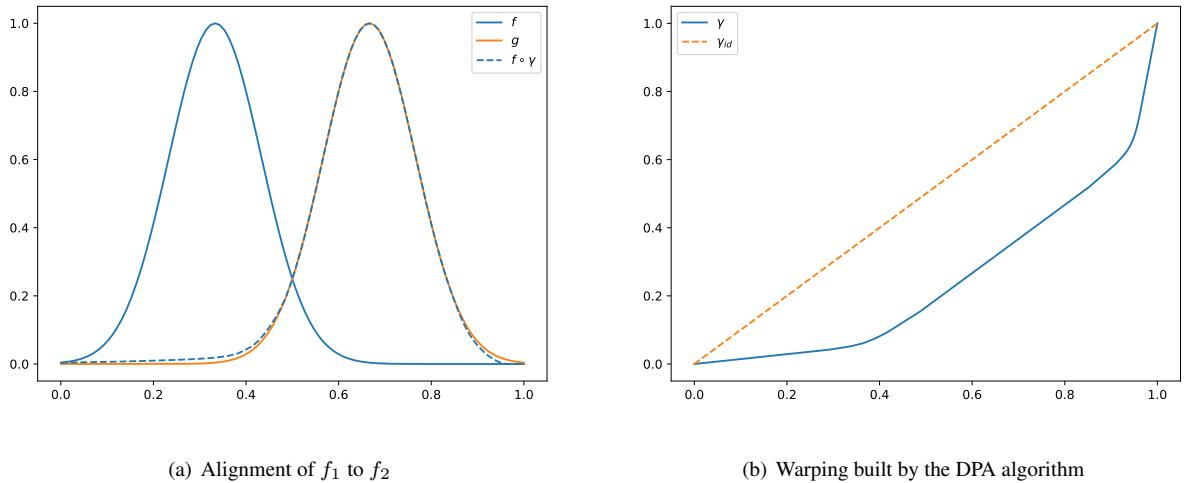


Figure 2.11: Pairwise alignment

To estimate γ^* , we will explore different paths that the reparameterization may take in a discretized grid, trying to minimize the energy term. In the appendix A.2 it is described in detail the algorithm used

for this task, which is usually referred by several authors as dynamic programing algorithm, or simply DPA [12], because it makes use of this programming technique to search the optimal path.

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 may be used as target.

A possible choice for the energy term would be to take $E[f, g] = \|f - g\|_{\mathbb{L}^2}^2$, but this criterion is not 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].

Pinching effect

It is a phenomenon we may encounter when we try to align two functions without a perfect match taking the \mathbb{L}^2 distance as energy. To minimize the term energy, the optimal solution will tend to squeeze an annoying region, until it disappears. Figure 2.12 shows the result of the registration of two functions with pinching effect. 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.

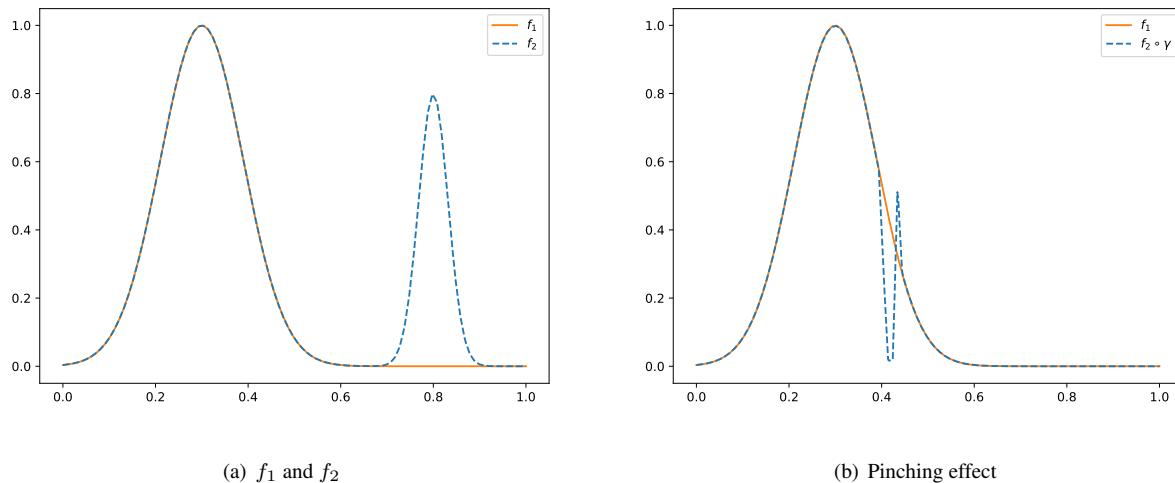


Figure 2.12: Pinching force effect

Lack of symmetry

Let $f_1, f_2 \in \mathcal{F}$ be two functions aligned. Generally, if we apply a reparameterization $\gamma \in \Gamma$ to both functions, their distance will change, thus

$$\|f_1 \circ \gamma - f_2 \circ \gamma\|^2 = \int_{\mathcal{T}} (f_1(\gamma(t)) - f_2(\gamma(t)))^2 dt = \int_{\mathcal{T}} |f_1(s) - f_2(s)|^2 \frac{1}{|\dot{\gamma}(\gamma^{-1}(s))|} ds \quad (s = \gamma(t)), \quad (2.6)$$

in other words, the action of Γ over \mathcal{F} under de \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 will be 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 $\{x_i\}_{i=1}^N$ be a set of functional observations, and $\{y_i\}_{i=1}^N = \{x_i \circ \gamma_i\}_{i=1}^N$ the corresponding registered observations. Also, we will denote as \bar{x} and \bar{y} the cross-sectional means. The Total Mean Square Error is defined as

$$\text{MSE}_{total} = \frac{1}{N} \sum_{i=1}^N \|x_i(t) - \bar{x}(t)\|^2 = \frac{1}{N} \sum_{i=1}^N \int [x_i(t) - \bar{x}(t)]^2 dt. \quad (2.7)$$

We can decompose the error in the part to each of the two sources of variability.

$$\text{MSE}_{amp} = C_R \frac{1}{N} \sum_{i=1}^N \int [y_i(t) - \bar{y}(t)]^2 dt \quad \text{MSE}_{phase} = \int [C_R \bar{y}^2(t) - \bar{x}^2(t)] dt \quad (2.8)$$

Where C_R is a constant related with the covariance between $\dot{\gamma}_i$ and y_i^2 . May be proved[*] that $\text{MSE}_{total} = \text{MSE}_{amp} + \text{MSE}_{phase}$.

From this separation we will construct a square multiple correlation index R^2 , which will indicate the proportion of the variation due to the phase explained by our registration process. This index will be defined as

$$R^2 = \frac{\text{MSE}_{phase}}{\text{MSE}_{total}}. \quad (2.9)$$

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, variability in functional data may come from the intrinsic 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 pre-processing 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 deforming the domain of the functions throughout the analysis.

In Srivastava et. al. (2011) [15] it is presented a novel framework to treat this approach using the Fisher-Rao metric. Although this framework covers a wide variety of topics, 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 *fdausrvf* [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 issues, which make its use 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 will 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 [Ref to fisher paper] 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 [18].

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.

In our case our 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 \mathbb{L}^2 . 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

$$\langle\langle v_1, v_2 \rangle\rangle_f = \frac{1}{4} \int_0^1 \dot{v}_1(t) \dot{v}_2(t) \frac{1}{|\dot{f}(t)|} dt, \quad (2.10)$$

where \dot{v} denotes the derivative.

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

$$L[\alpha] = \int_0^1 \langle \langle \dot{\alpha}(\tau), \dot{\alpha}(\tau) \rangle \rangle_{\alpha(\tau)} d\tau. \quad (2.11)$$

This will allow 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 ,

$$d_{FR}(f_1, f_2) = \inf_{\alpha: [0,1] \rightarrow \mathcal{F}, \alpha(0)=f_1, \alpha(1)=f_2} L[\alpha]. \quad (2.12)$$

Finding the geodesic path directly is computationally unapproachable, for this reason we will introduce the **square root slope function** (SRSF) transform, which will greatly simplify this computation.

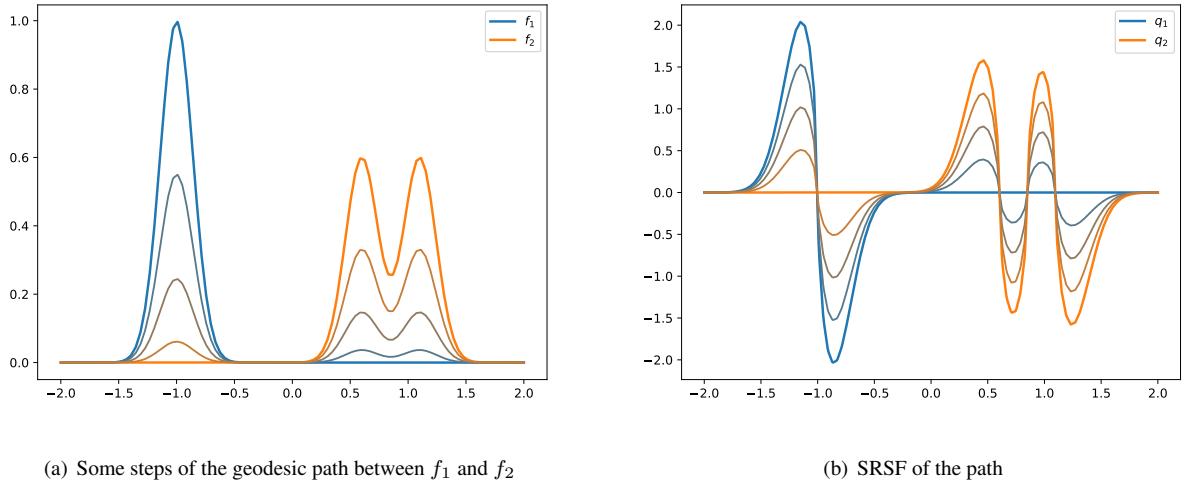


Figure 2.13: Geodesic path in \mathcal{F} and the corresponding SRSF's

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

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.4.

Taking advantage of this characterization we will take our functions to the SRSF's space to perform our analysis efficiently, and then transform the result to the original space, because the SRSF establishes a map up to constant between \mathcal{F} and the space of SRSF's with the \mathbb{L}^2 metric. A consequence is that the computation of geodesics shown in 2.13 becomes in a straight line between SRSFs,

$$\alpha(\tau) = (1 - \tau)q_1 + \tau q_2 \quad 0 \leq \tau \leq 1. \quad (2.13)$$

Given $\gamma \in \Gamma$, when it is calculated the SRSF of $f \circ \gamma$, due to the chain rule, we can observe that $SRSF\{f \circ \gamma\} = \text{sgn}(\dot{f} \circ \gamma) \sqrt{\dot{f} \circ \gamma} \sqrt{\dot{\gamma}} = (q \circ \gamma) \sqrt{\dot{\gamma}}$. To simplify the notation we will denote the SRSF of this composition as (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.4. Figure 2.14 shows a diagram with the action of the composition in the different spaces.

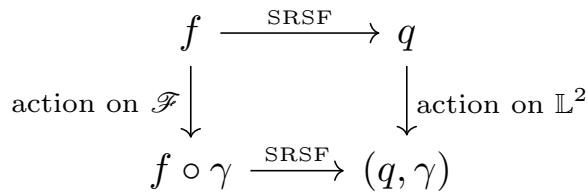


Figure 2.14: Action of Γ

2.3.2. Amplitude and phase spaces

We will formalize the amplitude and phase distance between functions of \mathcal{F} , which will come in a natural way due to the properties of the metric.

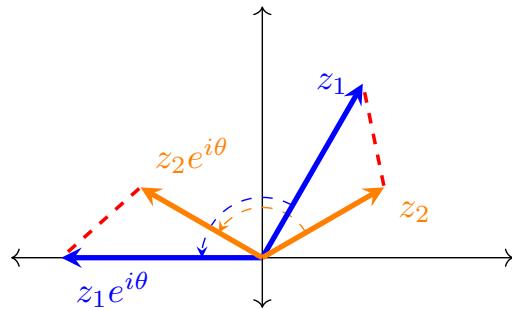


Figure 2.15: Rotation in complex plane

Firstly, before we proceed to define the phase and amplitude space, we will see an analogy with the rotations on the complex plane, which will be very useful to understand the role played by each of the magnitudes in our manifold.

Let z_1, z_2 be two points in \mathbb{C} , as it is illustrated in the figure 2.15, when it is applied a rotation on the plane, their distance remains invariant, i.e., it is an action by isometries, as the reparameterizations on our manifold.

The variability between two vectors in the plane can be 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.

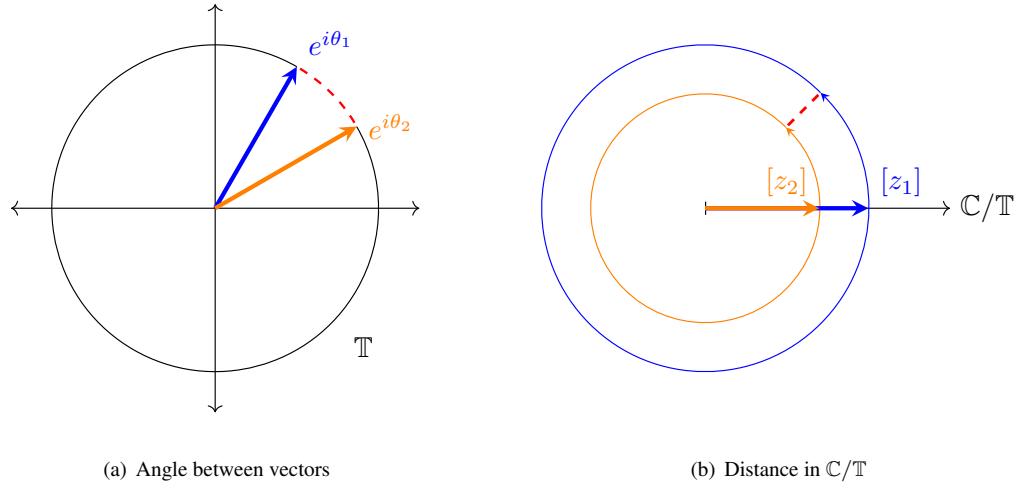


Figure 2.16: Phase and amplitude in the complex plane

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 2.17(a) it is shown some reparameterizations associated to a function of \mathcal{F} and their corresponding SRSF's in 2.17(b).

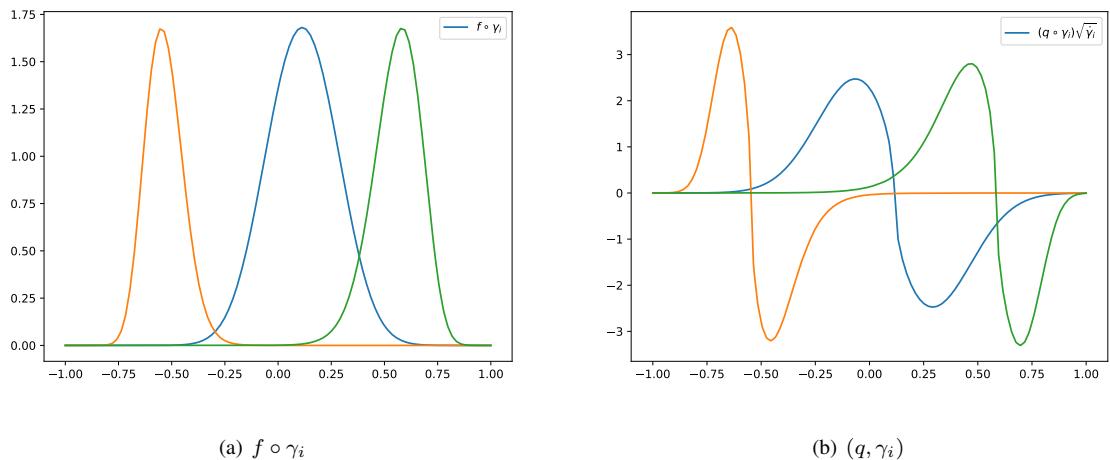


Figure 2.17: Functions in the same orbit

We will denote amplitude space $\mathcal{A} = \mathbb{L}^2/\Gamma$ 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

$$d_a([q_1], [q_2]) = \inf_{\gamma_1, \gamma_2 \in \Gamma} (\|(q_1, \gamma_1) - (q_2, \gamma_2)\|), \quad (2.14)$$

To quantify the other source of variability, we will define the phase space, which will be denoted as $\Gamma = \{\gamma : [0, 1] \rightarrow [0, 1] : \gamma \text{ is a boundary-preserving diffeomorphism}\}$, 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, their norm will be 1, thus

$$\|\gamma\|_\Gamma^2 = \|SRSF(\gamma)\|_{\mathbb{L}^2}^2 = \|\dot{\gamma}\|_{\mathbb{L}^2}^2 = \int_0^1 \sqrt{\dot{\gamma}(t)} \sqrt{\dot{\gamma}(t)} dt = \int_0^1 \dot{\gamma}(t) dt = \gamma(1) - \gamma(0) = 1. \quad (2.15)$$

The SRSF transform will be 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 use the structure of \mathbb{S}_∞ .

Let γ_1, γ_2 be in Γ and $\psi_1 = \sqrt{\dot{\gamma}_1}, \psi_2 = \sqrt{\dot{\gamma}_2}$ be their SRSF, their distance in Γ will be given by

$$d_{phase}(\gamma_1, \gamma_2) = d_\psi(\psi_1, \psi_2) = \cos^{-1} \left(\int_0^1 \psi_1(t) \psi_2(t) dt \right). \quad (2.16)$$

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

$$(\gamma_1^*, \gamma_2^*) = \operatorname{argmin}_{\gamma_1, \gamma_2 \in \Gamma} \|(q_1, \gamma_1) - (q_2, \gamma_2)\| \in \Gamma \times \Gamma. \quad (2.17)$$

Which will allow us to calculate the phase variability between them, as a generalization of the notion of an angle. The phase distance between these functions will be the distance of their relative phase $d_{phase}(\gamma_1^*, \gamma_2^*)$. Alternatively, we can use the properties of the metric to define the relative phase depending on a single warping, $\gamma_{12}^* = \operatorname{argmin}_{\gamma \in \Gamma} (\|(q_1, \gamma) - q_2\|)$, which is equivalent to set γ_2^* to be identity in 2.17.

2.3.3. Pairwise alignment

The Fisher-Rao metric will 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, i.e., the warping used in the alignment will be

$$\gamma^* = \operatorname{argmin}_{\gamma \in \Gamma} d_{FR}(f_1 \circ \gamma, f_2) = \operatorname{argmin}_{\gamma \in \Gamma} \|(q_1, \gamma) - q_2\|_{\mathbb{L}^2}. \quad (2.18)$$

The energy used to align the functions in figure 2.11(a) was actually this. We will have a property of symmetry, 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

$$E[f_1 \circ \gamma, f_2] = E[f_1 \circ \gamma \circ \gamma^{-1}, f_2 \circ \gamma^{-1}] = E[f_1, f_2 \circ \gamma^{-1}], \quad (2.19)$$

so that the warping needed to register f_2 to f_1 is γ^{*-1} .

In addition, the so-called pinching effect will not appear, due to the term $\sqrt{\gamma}$ 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. This mean minimizes the sum of square distances.

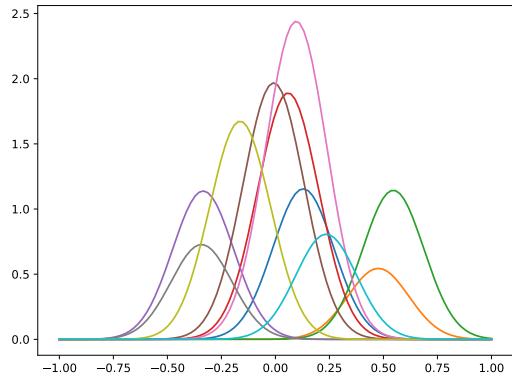
In the functional case, the mean function will have this property thus if $\{f_i\}_{i=1}^N \subset \mathbb{L}^2$ is a set of functional observations, their mean $\bar{f}(t) = \frac{1}{N} \sum_{i=1}^N f_i(t)$ will minimize $\sum_{i=1}^N \|f_i - \bar{f}\|_{\mathbb{L}^2}$.

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 $p \in X$ as $\Psi(p) = \sum_{i=1}^N d^2(p, x_i)$.

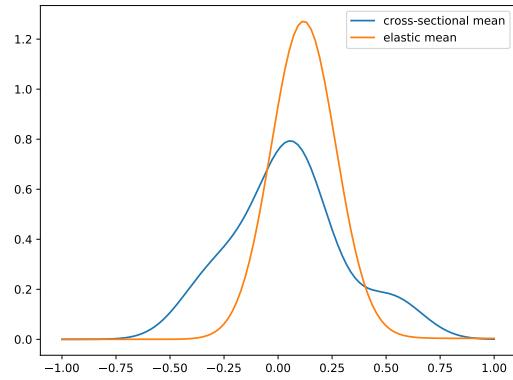
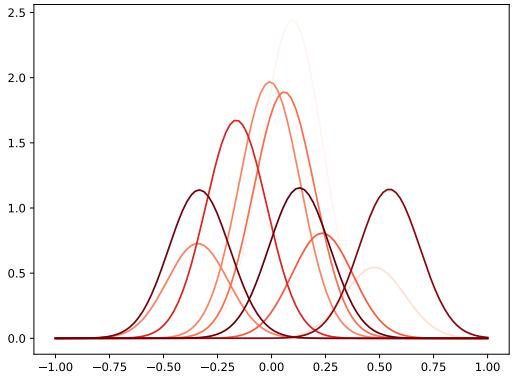
The Fréchet mean of these random points will be defined as the element $m \in X$ which globally minimizes the Fréchet Variance. If this global minimum does not exist, we will call Karcher means to the points that locally minimizes $\Psi(p)$, i. e.,

$$m = \operatorname{argmin}_{p \in M} \sum_{i=1}^N d^2(p, x_i) \quad (2.20)$$

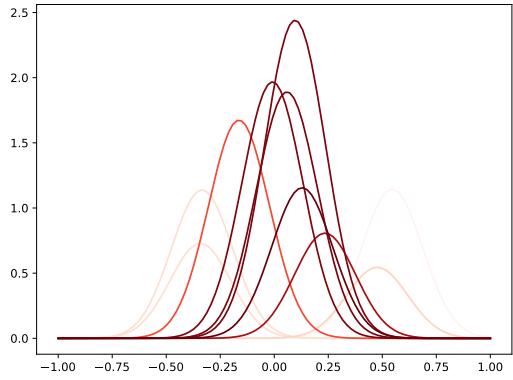
These Karcher means will be able to better capture the geometry of the problem than usual mean, as it is shown in the figure 2.18(b), where it is used a Karcher mean on \mathcal{A} . To calculate this mean it is used the DPA algorithm explained in the appendix A.2. For instance, the dataset shown in 2.18(a) contains gaussian-like samples, however the usual mean of figure 2.18(b) is not able to reflect this shape, unlike the Karcher mean in \mathcal{A} .



(a) Dataset of unimodal samples

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

(c) Ranked by elastic distance



(d) Ranked by phase distance

Figure 2.18: Karcher mean of dataset

As an example of the behavior of these means with different metrics, we may measure the centrality of an observation in a dataset using its distance to the Karcher mean. In the figure 2.18 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 2.18(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 2.18(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 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 \mathcal{A} , defined as

$$[\mu_q] = \underset{[q] \in \mathcal{A}}{\operatorname{arginf}} \sum_{i=1}^n d_a([q], [q_i])^2. \quad (2.21)$$

The bracket notation $[\mu_q]$ is used to emphasize the fact 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 relative phases γ_i^* between q_i and $[\mu_q]$ have as Karcher mean the identity.

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., $\gamma_i = \operatorname{arginf}_{\gamma \in \Gamma} \|\tilde{q} - (q_i, \gamma)\|$.

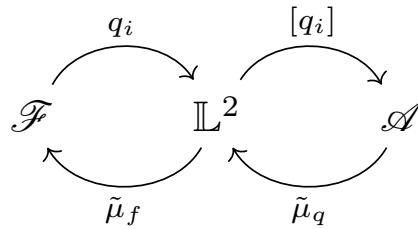


Figure 2.19: Scheme of the elastic registration procedure.

Then, we will compute the Karcher mean of $\{\gamma_i\}$, denoted as $\bar{\gamma}$, in the phase space, using the algorithm detailed in the appendix B.3. The center of the orbit will be $\tilde{\mu}_q = (\mu_q, \bar{\gamma}^{-1})$.

The mean of the relative phases with respect to $\tilde{\mu}_q$, will be the identity, thus

$$\operatorname{arginf}_{\gamma \in \Gamma} d_{FR}(\gamma_i \circ \bar{\gamma}^{-1}, \gamma) = \operatorname{arginf}_{\gamma \in \Gamma} d_{FR}(\gamma_i \circ \bar{\gamma}^{-1} \circ \bar{\gamma}, \gamma \circ \bar{\gamma}) = \operatorname{arginf}_{\gamma \in \Gamma} d_{FR}(\gamma_i, \gamma \circ \bar{\gamma}) = \gamma_{id}. \quad (2.22)$$

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 2.20 may be observed the result of the registration of the berkeley growth curves of the figure 2.6.

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

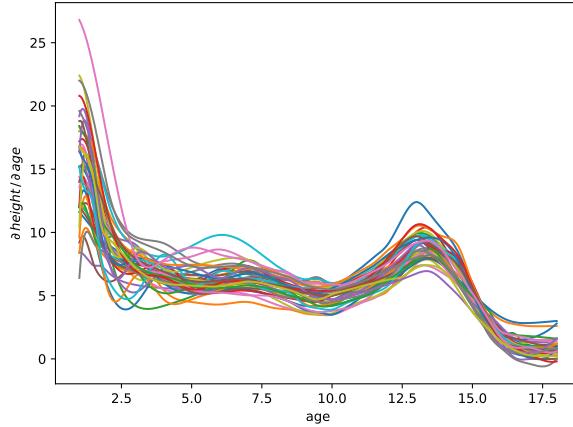


Figure 2.20: Elastic registration of the Berkeley velocity curves

$\{\gamma_i\}_{i=1}^n$ 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 [refff to paper].

2.3.6. Restricting elasticity

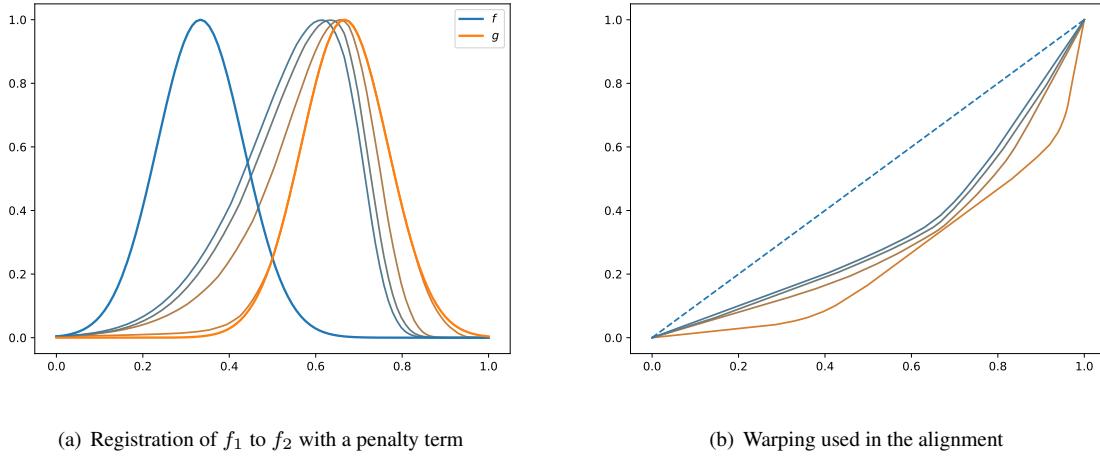


Figure 2.21: Penalized elastic registration with different values of λ

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 2.14. Let $q_1, q_2 \in \mathbb{L}^2$ be two SRSFs and $\lambda > 0$, we will define the penalized elastic distance as

$$d_\lambda(q_1, q_2) \equiv \inf_{\gamma \in \Gamma} \left(\|q_1 - (q_2 \circ \gamma \sqrt{\dot{\gamma}})\|^2 + \lambda \|\sqrt{\dot{\gamma}} - 1\|^2 \right)^{(1/2)}. \quad (2.23)$$

2.4. Nearest neighbors

The nearest neighbors estimators (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 a dataset.

Although in their classic version they are used with sets of vectors, their ideas will work in the same way in general metric spaces [19], as the functional ones we will work with.

2.4.1. Nearest neighbor search

Let (\mathcal{F}, d) be a metric space and $(\mathcal{X}_i, Y_i)_{i \leq i \leq n}$ 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 estimators, it is taken as neighborhood the k closest elements to x , i. e., if the training pairs are re-indexed as $(\mathcal{X}_{(i)}, Y_{(i)}) 1 \leq i \leq n$ so that the $\mathcal{X}_{(i)}$'s are re-arranged in increasing distance from x , $d(x, \mathcal{X}_{(1)}) \leq d(x, \mathcal{X}_{(2)}) \leq \dots \leq d(x, \mathcal{X}_{(n)})$, then $k(x) = \{\mathcal{X}_{(i)} : 1 \leq i \leq i \leq k\}$.

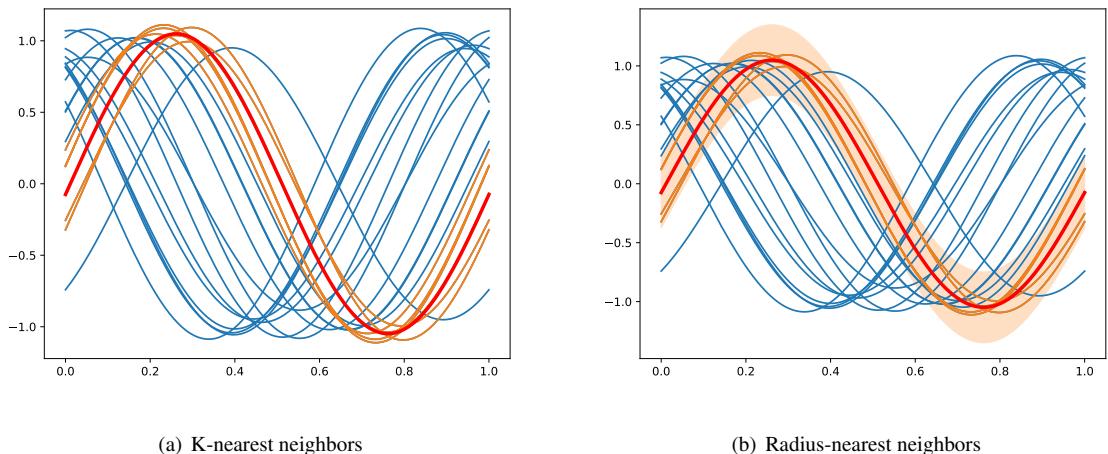


Figure 2.22: neighborhoods using distance \mathbb{L}^∞

In the second variant, less used in practice, consisting of the radius neighbors estimators, the neighborhood contains the samples \mathcal{X}_i in the ball of radius r centered in x , i.e. $k(x) = \{\mathcal{X}_i : d(\mathcal{X}_i, x) \leq r\}$. For instance, if we use the distance \mathbb{L}^∞ , we may visualize $k(x)$ as the set of all functions within a band of radius r around x .

In the figure 2.22 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 improved using data structures based on spatial indexes, such as ball trees [20]. However, the best nearest-neighbor data structure for a given application will depend on the dimensionality, size, and underlying structure of the data.

2.4.2. Classification

In the classification problem, the samples of the training set \mathcal{X}_i are associated to the labels Y_i corresponding to their class. Given a datum x , we will predict a new label using the majority class among its neighbors, so that the predicted class will be

$$\hat{Y} = \operatorname{argmax}_j \sum_{\mathcal{X}_i \in k(x)} \mathbb{1}_{\{Y_i=j\}}. \quad (2.24)$$

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

2.4.3. Regression

In the regression problem, each of the training samples \mathcal{X}_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 will be 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 will be used, so that the prediction will be calculated as

$$\hat{Y} = \sum_{(X_i, Y_i): X_i \in k(x)} w_i Y_i, \quad (2.25)$$

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

In the case where the responses \mathcal{Y}_i are also functional data, the predicted response will be 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.

DESIGN AND DEVELOPMENT

So far, we have focused on the mathematical framework of the functionalities incorporated into the package, however, most of the work was done during the design, development and integration parts.

Although the package is currently at an early stage of development, it is already beginning to be used by researchers in different countries, for this reason, and to allow the expansion of the project, it has not only been necessary the coordination of the team involved, but also special emphasis has been placed on carrying out the development in a public and transparent way and making a responsible integration of the new functionalities.



Figure 3.1: scikit-fda logo

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 an extensive test bench of unit test and continuous integration mechanism.

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

- 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 in the different Python versions supported.
- API should be similar, as far as possible, to the numpy [21], scipy [22] and scikit-learn [23] 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 FDataBase, 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 with a high-level description of this structure. The following subsections summarize, in general terms, the functionalities incorporated and design changes made during this work. A more detailed description may be found in Annex C, or in the documentation available online.

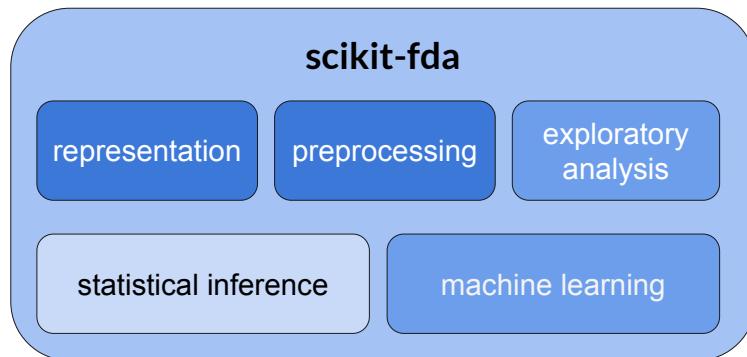


Figure 3.2: Map of scikit-fda [24]

3.2.1. Representation module

One of the most important changes carried out has been the creation of the FData class, from which the existing FDataGrid and FDataBase inherit, designed to unify the API of the different representations. This class implements general methods for evaluation, extrapolation, and plotting and defines abstract ones common to all representations.

In addition, two submodules have been added, *interpolation* and *extrapolation*, containing classes

used during the evaluation of functional data, implementing the concepts explained in section 2.1.

3.2.2. Preprocessing module

In the preprocessing module a specific sub-module was created to deal with the registration problem, with methods to deal with the data with the techniques studied throughout sections 2.2 and 2.3, allowing data to be registered by means of shifts, landmarks or the elastic approach.

3.2.3. Machine learning module

The machine learning module is divided into three submodules: classification, regression and clustering. Classes have been added for the classification and regression of functional data with estimators based on nearest neighbors explained in section 2.4.

3.2.4. Miscellaneous module

This module contains different scattered topics used during the analysis of functional data. Specifically, it has been added a metrics submodule, which contains functional metrics, among which there are the family of L^p metrics for multivariate functions, the Fisher-Rao distance or elastic metrics based on the framework of section 2.3.2.

3.2.5. Dataset module

In the dataset module have been included synthetic generators, used during the development of tests or examples. May be found generators of sinusoidal processes, multimodal samples or random warpings.

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.

In addition, because of the software-free character 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 Proposals (PEPs),

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. These pages generated automatically are maintained by the continuous integration system and may be consulted online on <https://fda.readthedocs.io/>.

The figure shows two parts of the Scikit-fda online documentation:

(a) Documentation of function

```
skfda.preprocessing.registration.invert_warping(fdatagrid, *, eval_points=None) [source]
```
Compute the inverse of a diffeomorphism.

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

```

(b) Doctest included within the documentation

```
Examples
```
>>> import numpy as np
>>> from skfda import FDataGrid
>>> from skfda.preprocessing.registration import invert_warping

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

>>> 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 3.3: 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 de documentation, as it is shown in the figure 3.3(b).

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.

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. Below it is shown the beginning of one of these examples.

HERE WILL BE AN EXAMPLE.

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 [25], which is based on simplicity in development, continuous feedback and communication between team members.

For this communication in conjunction with version control git has been using, along with the functionalities available in Github. 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 3.4 illustrates the mechanics of this workflow.

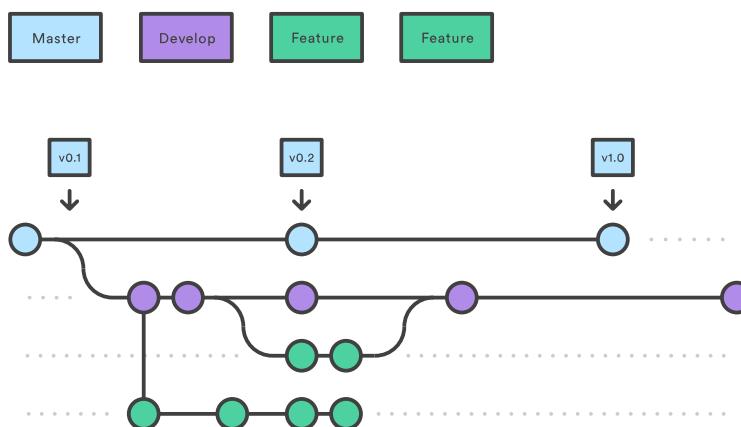


Figure 3.4: Example of git flow branches ¹

One of the main advantages of using github, over other version control systems, is the possibility of using a continuous integration mechanism. Travis CI has been used for this purpose, which compiles the library and runs the bench tests each time a commit is made, as in figure 3.5, where it may be seen a red cross or a green tick associated to each commit depending on the bench tests. In addition, it can also automate a wide variety of tasks, such as generating the documentation, compiling or integrating style reviewers.

¹ Picture from <https://es.atlassian.com/git>, licensed under a Creative Commons Attribution 2.5 Australia License



Figure 3.5: Example of travis checks

CONCLUSIONS AND FUTURE WORK

The goal of this project is the creation of a Python reference package in FDA, along with the creation of a community to support and keep the project updated. So it will require the effort and dedication of a team of developers to achieve this goal.

During this year, the project has evolved drastically, redesigning and expanding its functionalities. 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. Among these functionalities that it would be interesting to cover the basis representations of functional data, including support for surfaces, so that covariances can be represented in this form, and to support more types of basis, such as wavelets [26] or constrained basis [7].

It would also be interesting to extend 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, to discover the library to interested developers and researchers, presenting the package in congresses and publishing articles to promote its use.

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 ;," *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*. PhD thesis, Universidad Autónoma de Madrid, 2018.
- [6] "III International Workshop on Advances in Functional Data Analysis," 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] C. R. de Boor, "B(asic)-Spline Basics," vol. 3, no. September, 1981.
- [10] H. E. Jone and N. Bayley, "The Berkeley Growth Study," *Child Development*, vol. 12, no. Jun, pp. 167–173, 1941.
- [11] P. Kokoszka and M. Reimherr, *Introduction to Functional Data Analysis*. Taylor & Francis Group, 2017.
- [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. Cencov, "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.
- [22] E. Jones, T. Oliphant, and P. Peterson, “{SciPy}: Open source scientific tools for {Python}.”
- [23] 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.
- [24] C. Ramos, “scikit-fda,” in *III International Workshop on Advances in Functional Data Analysis*, 2019.
- [25] J. Blankenship, M. Bussa, and S. Millet, “eXtreme Programming,” in *Pro Agile .NET Development with Scrum*, ch. 3, 2011.
- [26] P. A. Morettin, A. Pinheiro, and B. Vidakovic, *Wavelets in Functional Data Analysis*. Springer, 2017.

APPENDICES

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 $\{x_i\}_{i=1}^n$, according to ??, we will use a variation of the Newton-Raphson's root-finding algorithm, applied to the derivative of REGSSE (2.3). This procedure is explained in more detail in Ramsay and Silverman (2005) [7].

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

$$\frac{\partial}{\partial \delta_i} \text{REGSSE} = 2 \int_{\mathcal{T}} [x_i(t + \delta_i) - \hat{\mu}(t)] D x_i(t) dt, \quad (\text{A.1})$$

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

In practice the first term of $\frac{\partial^2}{\partial \delta_i^2} \text{REGSSE}$ (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:

$$\frac{\partial^2}{\partial \delta_i^2} \text{REGSSE} \approx 2 \int_{\mathcal{T}} [D x_i(t)^2] dt. \quad (\text{A.3})$$

The initialization 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 maximum 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  $\{x_i(t)\}_{i=1}^n$ 
Output: Shifts  $\{\delta_i\}_{i=1}^n$  used to register the data

1 Initialize  $\delta^{(0)}$  ;
2 for step  $\nu = 1, 2, \dots$  until stop criterion do
3   Update cross-sectional mean
4    $\hat{\mu}(t) \leftarrow \frac{1}{n} \sum_{i=1}^n x_i(t + \delta_i^{(\nu-1)})$ 
5   foreach  $\delta_i^{(\nu-1)}$  do
6     Update values of  $\delta_i$ 
7      $\delta_i^{(\nu)} \leftarrow \delta_i^{(\nu-1)} - \alpha \frac{\partial}{\partial \delta_i} REGSSE / \frac{\partial^2}{\partial \delta_i^2} REGSSE$  ;
8   end

```

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

A.2. Dynamic programming algorithm

In order to find the optimal reparameterization between two functions, according to eqn. 2.18 we will use a dynamic programming algorithm, see [8] for more details.

Para el computo de γ es una difeomorfismo en $[0, 1] \rightarrow [0, 1]$, por tanto lo podemos entender como un camino estrictamente creciente en el rectángulo $[0, 1] \times [0, 1]$, el cual aproximaderos por una malla discreta de valores $G_N \times G_N$ con $G_N = \{(i/N)\}_{i=0}^N$. Con esta aproximación estaremos aproximando γ como una función lineal a trozos.

En la práctica no es necesario explorar todos los valores, nos podemos limitar a una partición. Por ejemplo en la figura que puede tomar un warping a partir de cierto punto.

We need to find a warping function γ which gives a path in

A.3. Karcher means computation

A.4. 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: $f(t) \rightarrow \dot{f}(t) \rightarrow q(t) = \text{sign}(\dot{f})\sqrt{|\dot{f}|} = Q(\dot{f})$.

For any $v \in T_f(\mathcal{F})$, the differential of this mapping is $v(t) \rightarrow \dot{v}(t) \rightarrow w(t) = Q_{*,f(t)}(\dot{v}(t))$.

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:

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

The RHS is compared with Eqn.2.10 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 an arbitrary warping and $q1, q2 \in \mathbb{L}^2$,

$$\begin{aligned} \|(q_1, \gamma) - (q_2, \gamma)\|^2 &= \int_0^1 \left(q_1(\gamma(t)) \sqrt{\dot{\gamma}(t)} - q_2(\gamma(t)) \sqrt{\dot{\gamma}(t)} \right)^2 dt = \\ &\int_0^1 (q_1(\gamma(t)) - q_2(\gamma(t)))^2 \dot{\gamma}(t) dt = \|q_1 - q_2\|^2. \square \end{aligned}$$

EXAMPLE NOTEBOOKS

Introducción a las sección de ejemplos.

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

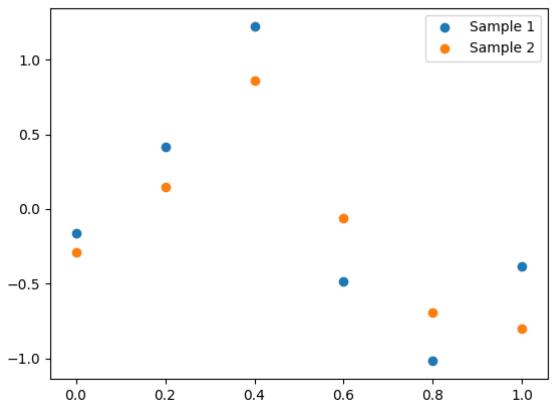
# sphinx_gallery_thumbnail_number = 3

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.

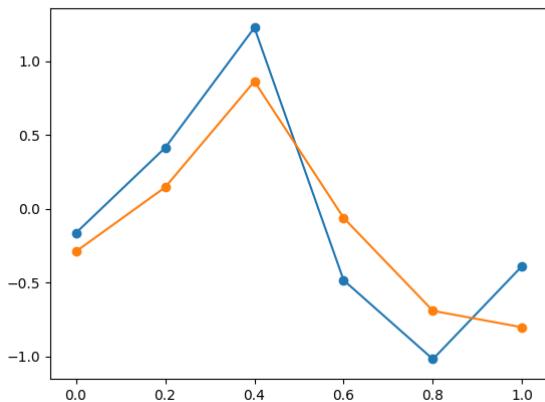
We will construct an example dataset with two curves with 6 points of discretization.

```
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()
```

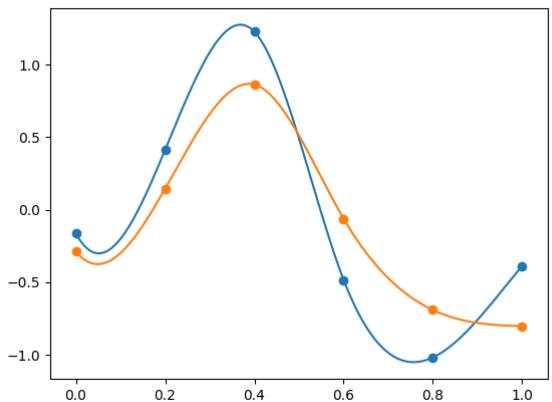


The interpolation method of the FDataGrid could be changed setting the attribute `interpolator`. Once we have set an interpolator it is used for the evaluation of the object.

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)

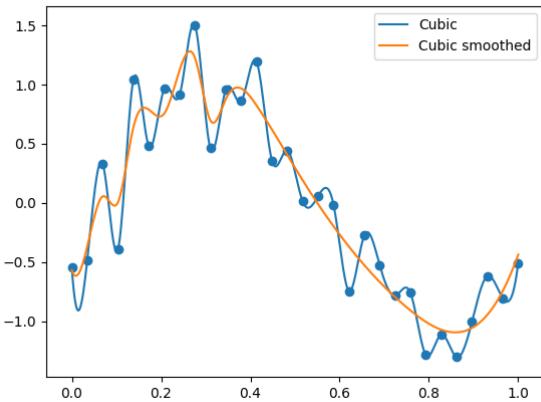
# Cubic interpolator
fd_smooth.interpolator = SplineInterpolator(interpolation_order=3)

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()
```

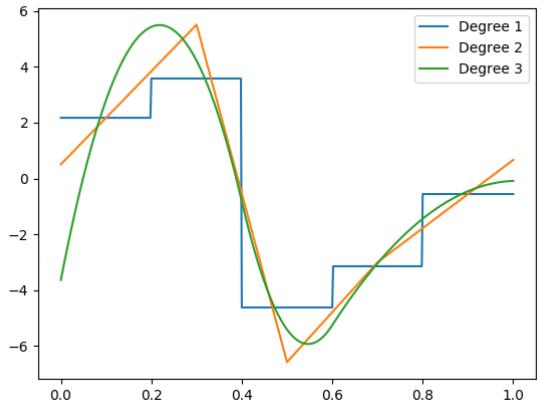


It is possible to evaluate derivatives of the FDataGrid, but due to the fact that interpolation is performed first, the interpolation loses one degree for each order of derivation. In the next example, it is shown the first derivative of a sample using interpolation with different degrees.

```
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()
```

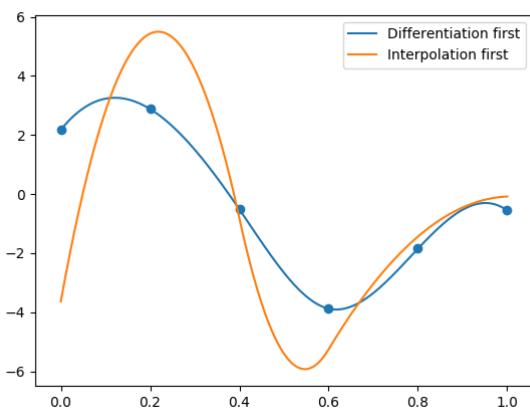


FDataGrids can be differentiate using lagged differences with the method `derivative()`, creating another FDataGrid which could be interpolated in order to avoid interpolating before differentiating.

```
fd_derivative = fd.derivative()
fd_derivative.plot(label="Differentiation first")
fd_derivative.scatter()

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

plt.legend()
```

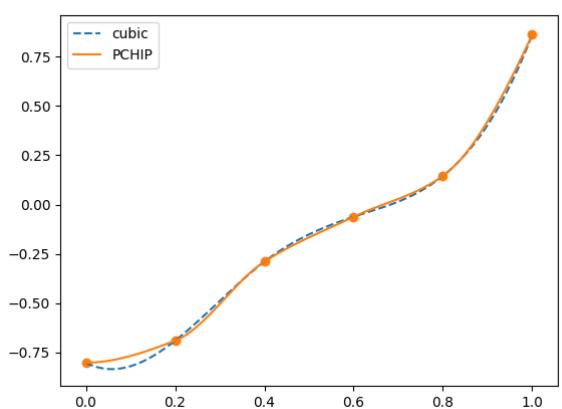


Sometimes our samples are required to be monotone, in these cases it is possible to use monotone cubic interpolation with the attribute `monotone`. A piecewise cubic hermite interpolating polynomial (PCHIP) will be used.

```
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()
```



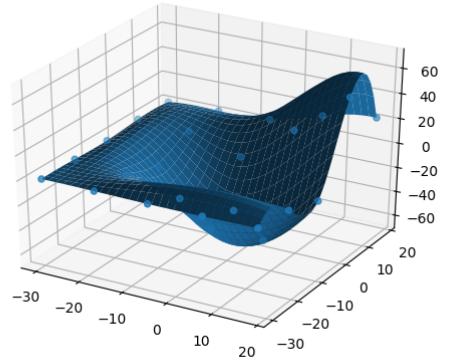
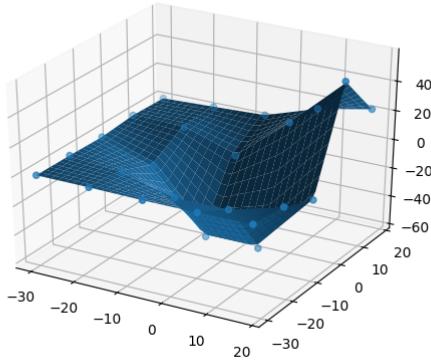
All the interpolators will work regardless of the dimension of the image, but depending on the domain dimension some methods will not be available.

For the next examples it is constructed a surface, $x_i : \mathbb{R}^2 \mapsto \mathbb{R}$. 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[:, :], 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)

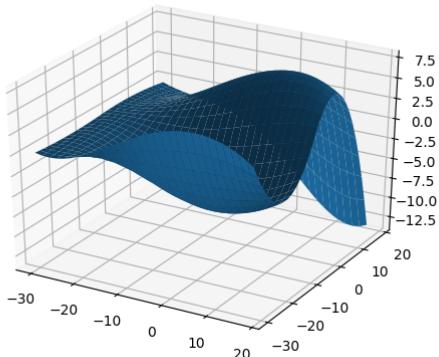
fig, ax = fd.plot()
fd.scatter(ax=ax)

plt.show()
```

In case of surface derivatives could be taken 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](#)

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
```

```
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 different 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 = ""
```

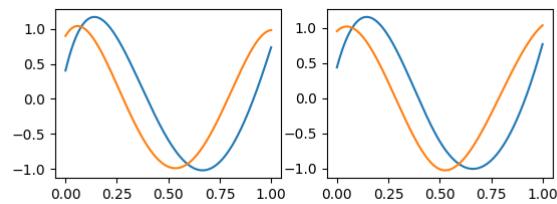
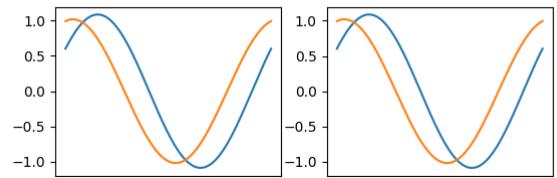
The extrapolation defines how to evaluate points that are outside the domain range of a `FDataBasis` or a `FDataGrid`.

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.

BSpline Basis



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_ylimits((-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.

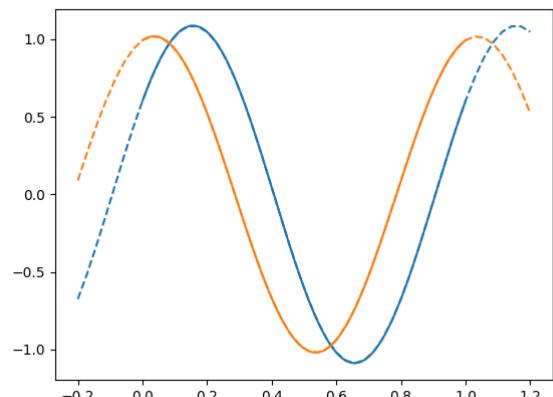
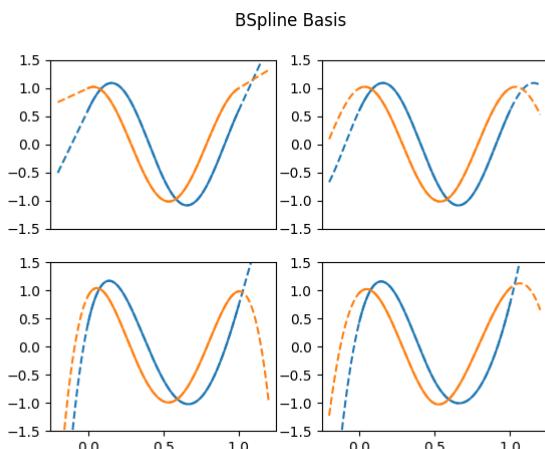
```
t = np.linspace("domain_extended")

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
```

Periodic extrapolation



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"

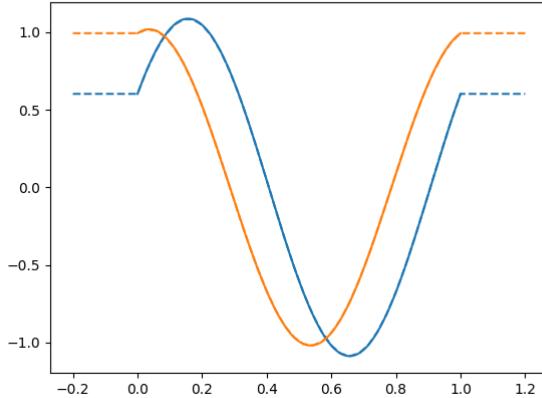
# Other way to call the extrapolation, changing the default value
fdgrid.extrapolation = "bounds"

# 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

```

Boundary extrapolation



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)`.

```

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

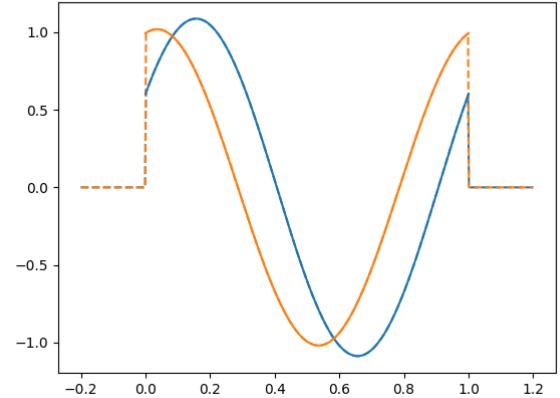
# Evaluation of the grid filling with zeros
fdgrid.extrapolation = "zeros"

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

plt.gca().set_prop_cycle(None) # Reset color cycle
fdgrid.plot() # Plot dataset

```

Fill with zeros



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.

```

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')

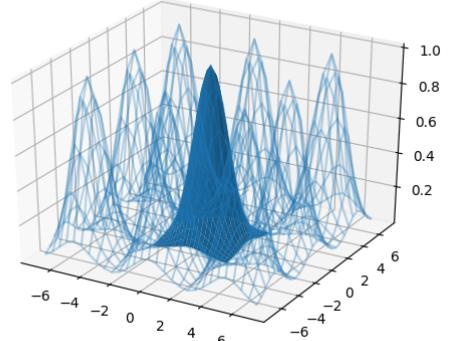
# Make data.
t = np.arange(-2.5, 2.75, 0.25)
X, Y = np.meshgrid(t, t)
Z = np.exp(-0.5 * (X**2 + Y**2))

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

T, S = np.meshgrid(t, t)

# Evaluation with periodic extrapolation
values = fd_surface((t,t), grid=True, extrapolation="periodic")
ax.plot_wireframe(T, S, values[0], alpha=.3, color="C0")
ax.plot_surface(X, Y, Z, color="C0")

```



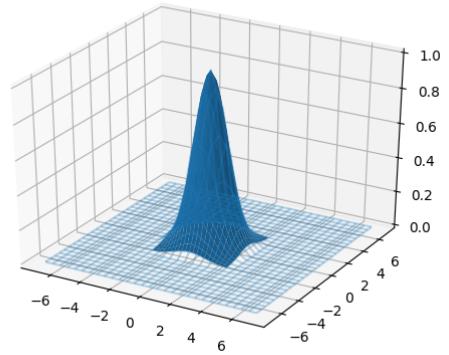
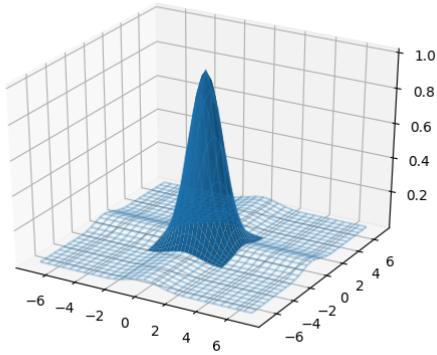
The previous extension can be compared with the extrapolation using the values of the bounds.

```

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

[↑][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 $f : X \rightarrow Y$ and $g : Y \rightarrow Z$, the composition will produce a third function $g \circ f : X \rightarrow Z$ 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.

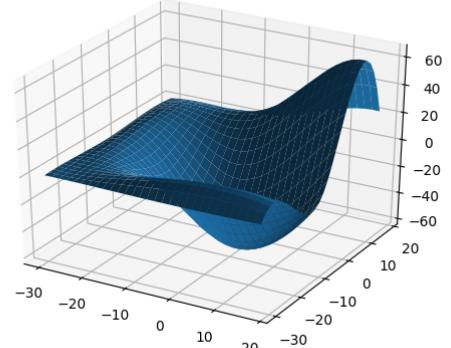
Firstly we will create a data object containing a surface $g : \mathbb{R}^2 \rightarrow \mathbb{R}$.

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

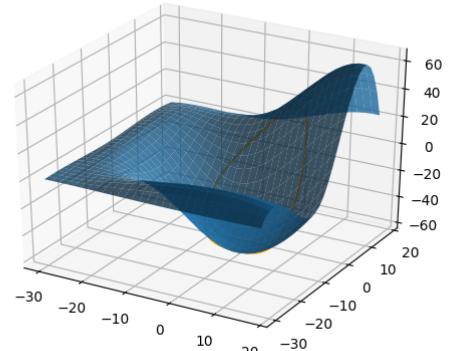
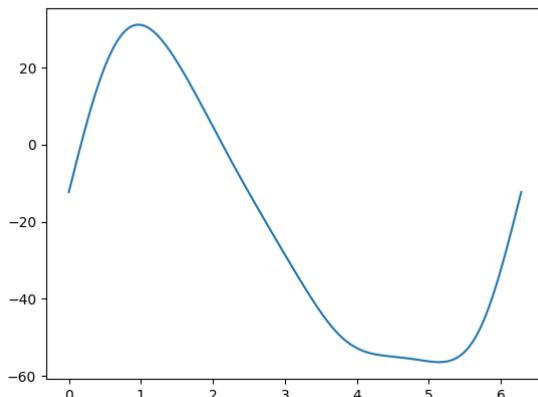


```
# Creation of circumference in parametric form
t = np.linspace(0, 2*np.pi, 100)

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()
```



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

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

[Download Python source code: plot_composition.py](#)

[Download Jupyter notebook: plot_composition.ipynb](#)

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

Gallery generated by Sphinx-Gallery

```
# 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()
```

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

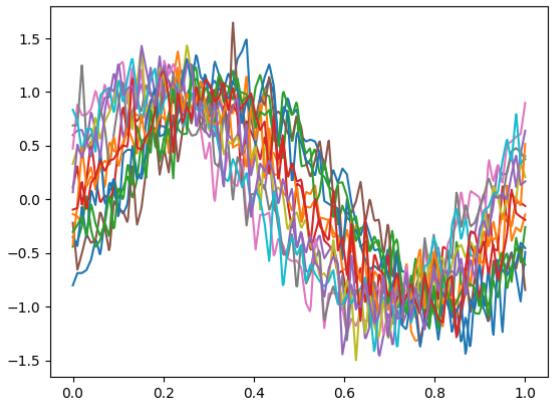
# sphinx_gallery_thumbnail_number = 3

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 contains 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()
```

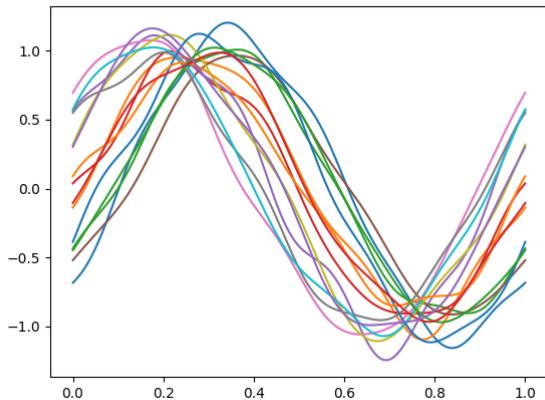


We will smooth the curves using a basis representation, which will help us to remove the gaussian noise. Smoothing before registration is essential due to the use of derivatives in the optimization process.

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()
```

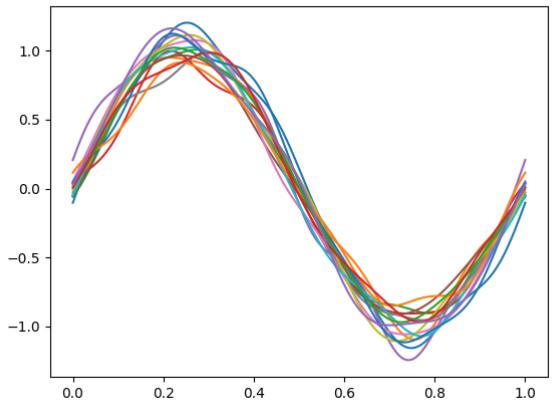


We will apply the `shift registration`, which is suitable due to the periodicity of the dataset and the small amount of amplitude variation.

```
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.

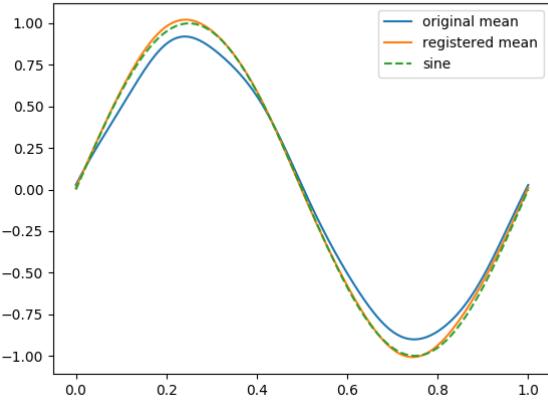
```
plt.figure()

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.

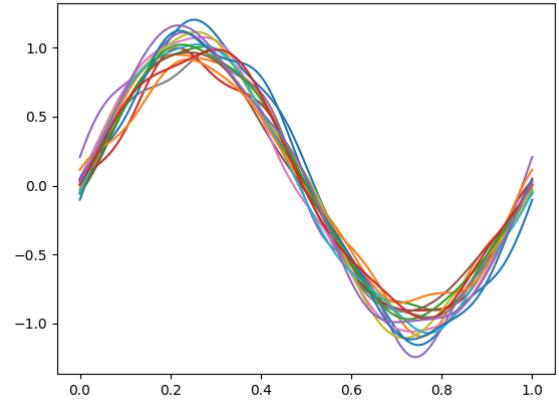
```
deltas = skfda.preprocessing.registration.shift_registration_deltas(fd_basis)
print(deltas)
```

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

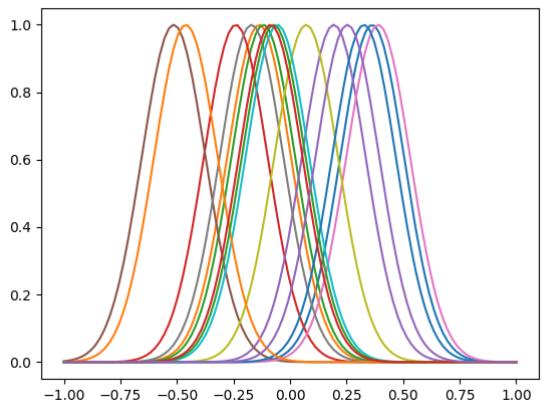
# sphinx_gallery_thumbnail_number = 2

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

We will use an example dataset synthetically generated by `make_multimodal_samples`, which 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()
```



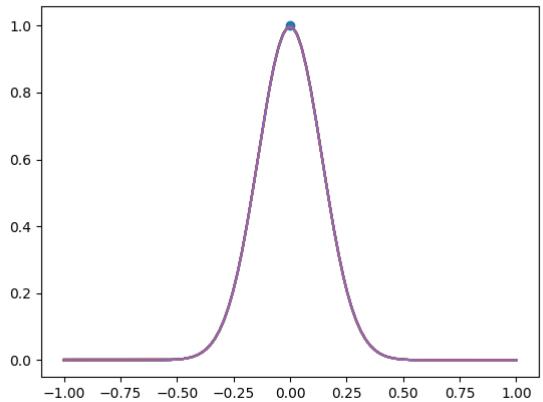
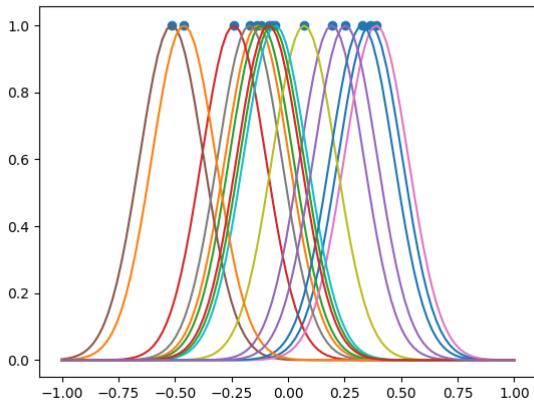
A landmark or a feature of a curve is some characteristic that one can associate with a specific argument value t . These are typically maxima, minima, or zero crossings of curves, and may be identified at the level of some derivatives as well as at the level of the curves themselves. [1]

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)
```

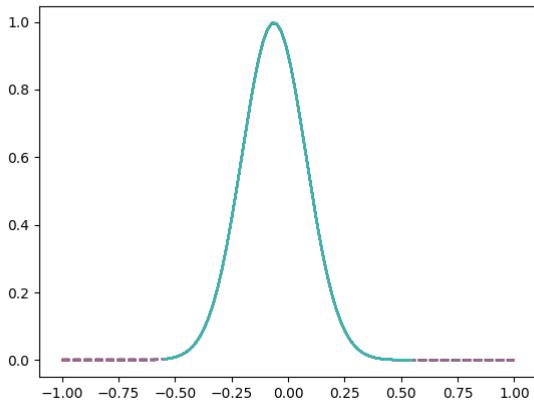
In many circumstances it is possible that we could not apply extrapolation, in these cases it is possible to restrict the domain to avoid evaluating points outside where our curves are defined.

If the location of the new reference point is not specified it is chosen the point that minimizes the maximum 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)

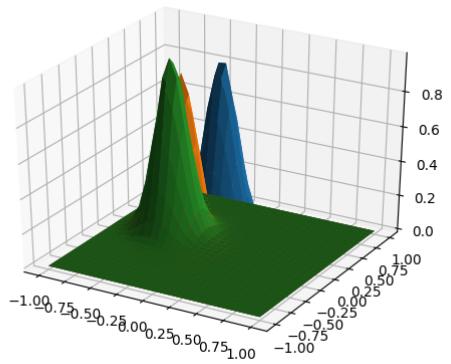
plt.figure()
l1 = fd_extrapolated.plot(linestyle='dashed', label='Extrapolated samples')
l2 = fd_restricted.plot(label="Restricted samples")
# plt.legend(handles=[l1[-1], l2[-1]])
```



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 dataset 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:

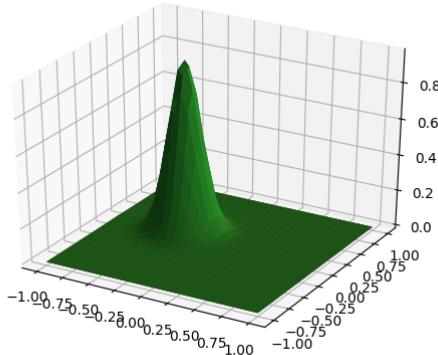
```
[[ 0.36321467 -0.13679289]
 [-0.11810279 -0.23992308]
 [ 0.19351103 -0.5146397 ]]
```

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](#)

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

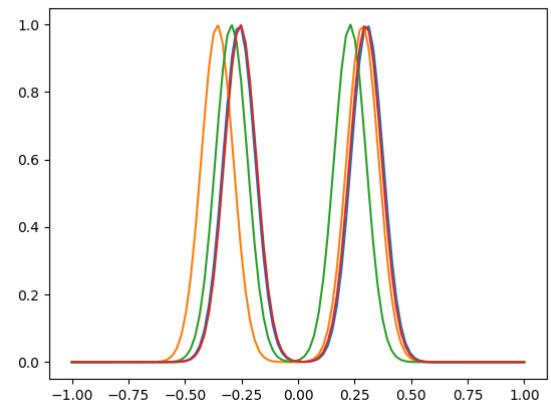
import skfda
import matplotlib.pyplot as plt
import numpy as np
```

The simplest curve alignment procedure is landmark registration. This method only takes into account a discrete amount 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`, which 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 maximum 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:

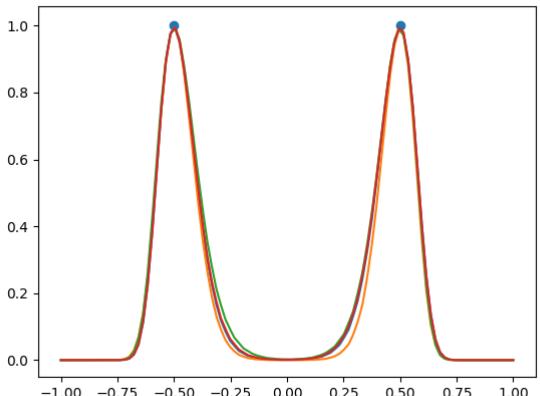
```
[[ -0.2606904  0.30597475]
 [-0.35695389  0.28534872]
 [-0.29463113  0.23040539]
 [-0.25530298  0.29929113]]
```

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

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])

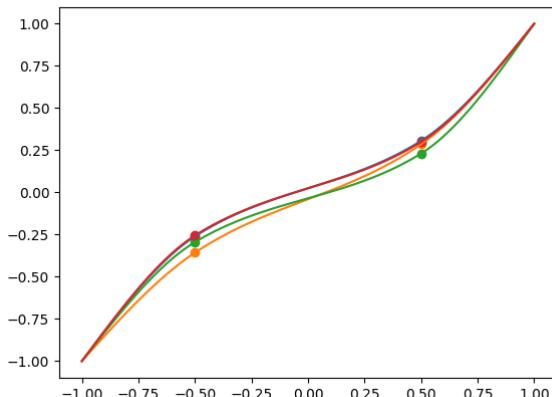
plt.show()
```

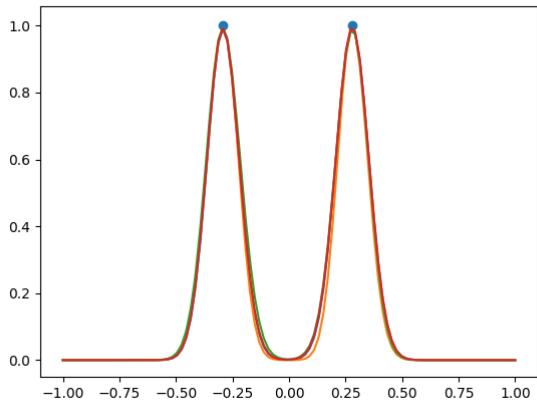
```
warping = skfda.preprocessing.registration.landmark_registration_warping(fd, landmarks,
                                                                      location=[-0.5, 0.5])

plt.figure()

# Plots warping
warping.plot()

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





- [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

NoteClick [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.

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

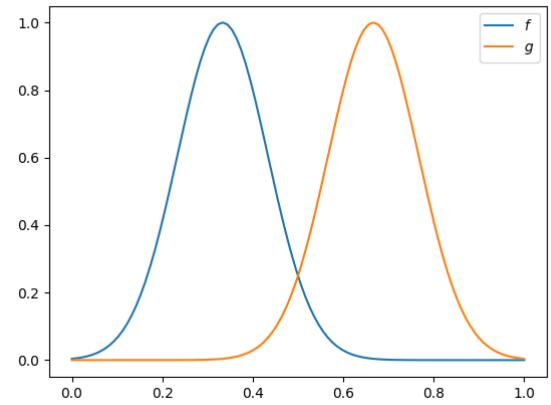
In the case of elastic registration it is taken as energy function the Fisher-Rao distance with a penalisation term, due to the property of invariance to reparameterizations of warpings functions.

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

Firstly, we will create two unimodal samples, f and g , defined in $[0, 1]$ which 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(['$f$', '$g$'])
```



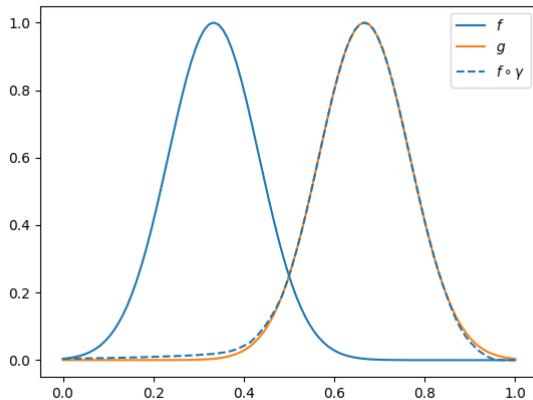
In this example g will be used as template and f will be aligned to it. In the following figure it is shown the result of the registration process, which can be computed using `elastic_registration`.

```
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`.

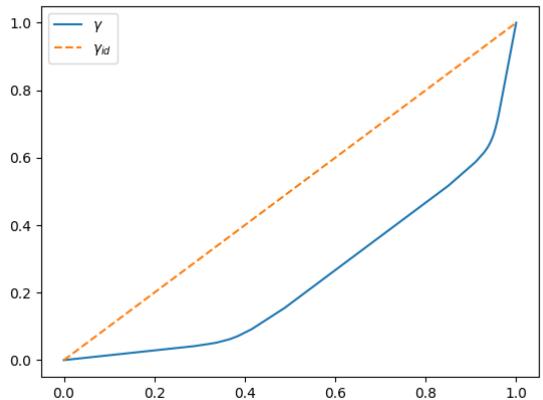
```
# Warping to align f to g
warping = skfda.preprocessing.registration.elastic_registration_warping(f, g)

plt.figure()

# Warping used
warping.plot()

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

# Legend
plt.legend(['$\gamma$', '$\gamma_d$'])
```



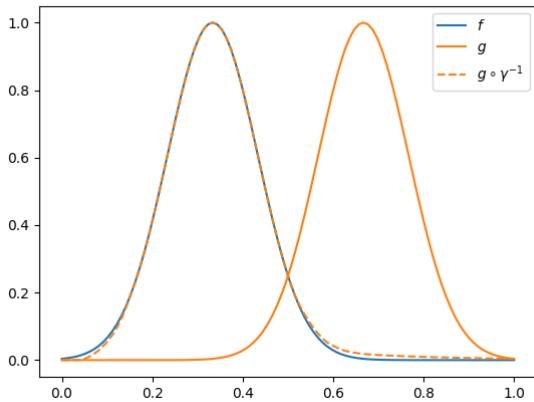
The transformation necessary to align g to f will be the inverse of the original warping function, γ^{-1} . This fact is a consequence of the use of the Fisher-Rao metric as energy function.

```
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 $\lambda \mathcal{R}(\gamma)$ which 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)

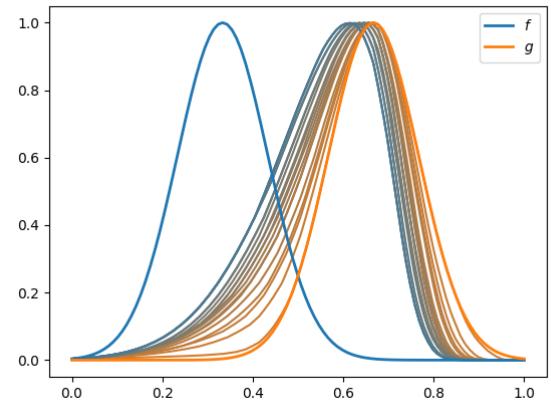
# Creation of a color gradient
cmap = plt.LinearSegmentedColormap.from_list('custom cmap', ['C1', 'C0'])
color = cmap(.2 + 3*lambdas)

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()
```

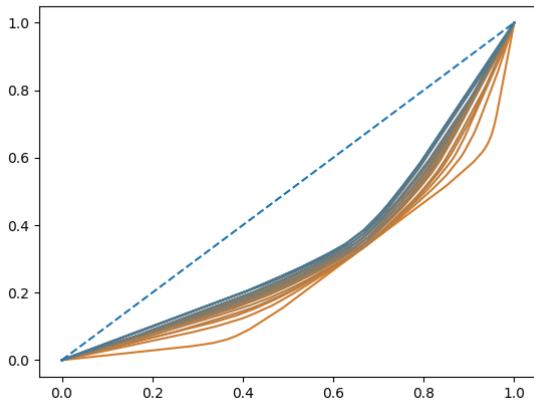


This phenomenon of loss of elasticity is clearly observed in the warpings used, since as the term of penalty increases, the functions are closer to γ_{id} .

```
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 alignment 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)

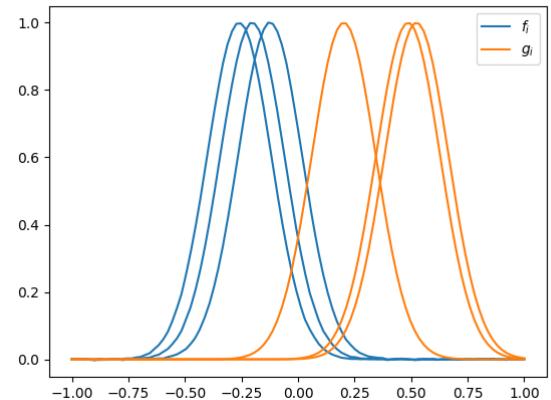
location1 = state.normal(loc=-.3, scale=.1, size=3)
fd = skfda.datasets.make_multimodal_samples(n_samples=3, modes_location=location1,
                                             noise=.001, random_state=1)

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="$f_i$")
g.plot(color='C1', label="$g_i$")

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



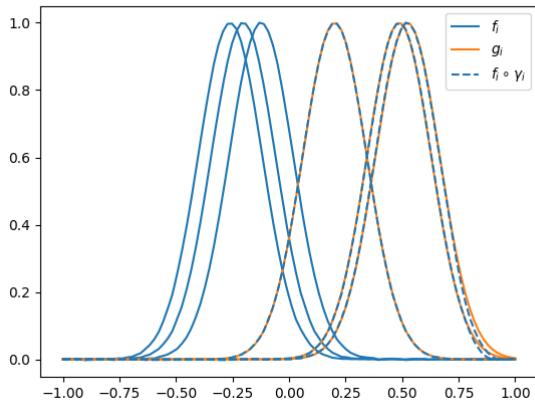
The following figure shows the result of the pairwise alignment of $\{f_i\}$ to $\{g_i\}$.

```
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

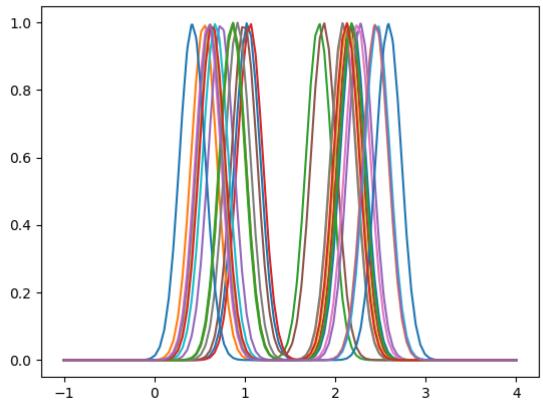
import skfda
import matplotlib.pyplot as plt
import numpy as np
```

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.

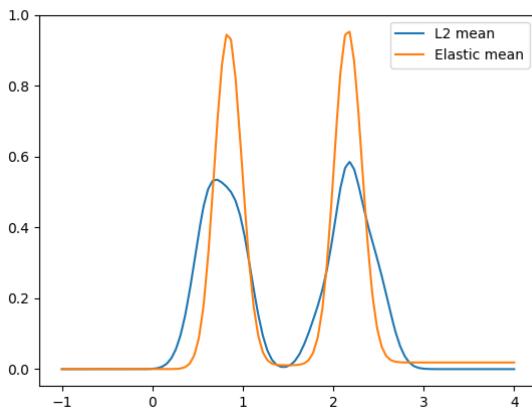
```
fd = skfda.datasets.make_multimodal_samples(n_modes=2, stop=4, random_state=1)
fd.plot()
```



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.

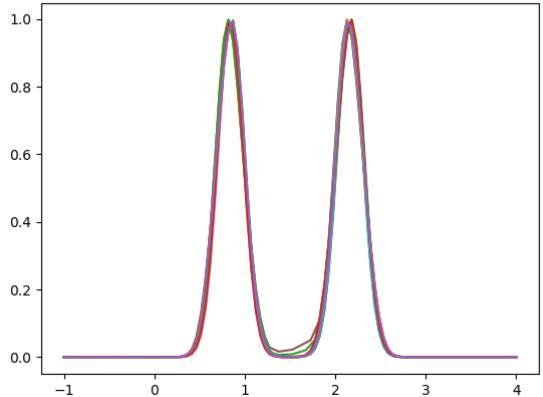
```
plt.figure()
fd.mean().plot(label="L2 mean")
skfda.preprocessing.registration.elastic_mean(fd).plot(label="Elastic mean")
plt.legend()
```



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 of children, in this case we will use only the males. The growth curves will be resampled using cubic interpolation and derived to obtain the velocity curves.

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

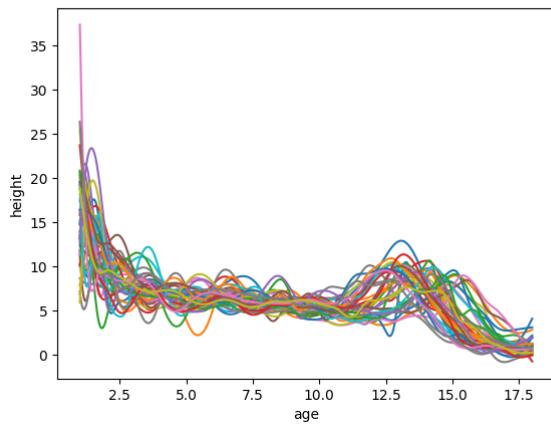
# Select only one sex
fd = growth['data'][growth['target'] == 0]

# 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"

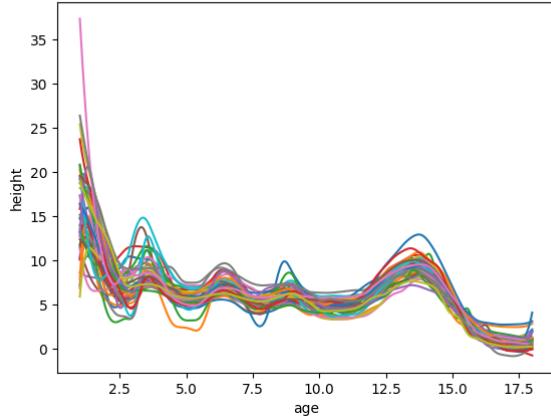
fd_align.plot()

plt.show()
```

[Download Python source code: plot_elastic_registration.py](#)[Download Jupyter notebook: plot_elastic_registration.ipynb](#)

Gallery generated by Sphinx-Gallery

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

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.

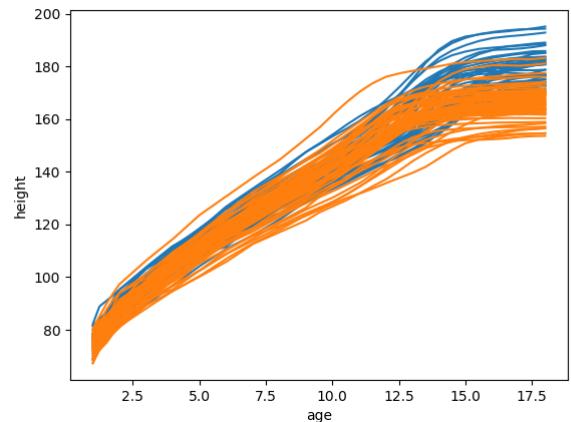
Firstly, we are going to fetch a functional data dataset, such as the Berkeley Growth Study. This dataset correspond to the height of several boys and girls measured until the 18 years of age.

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')
```



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 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 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)
```

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 assign the majority class. By default, it is used the 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 1 1 0 0 1 1 0 1 0 0 1 1 1 1 0 1]
```

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

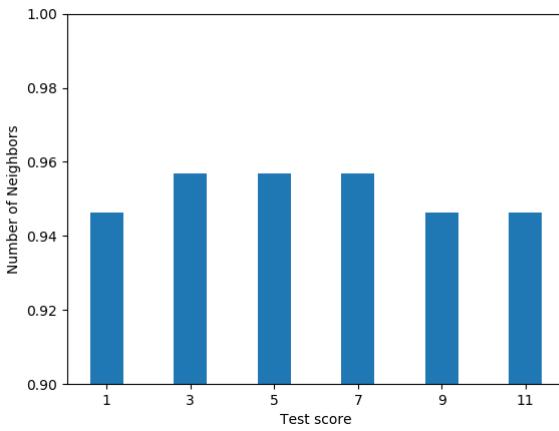
```
score = knn.score(X_test, y_test)
print(score)
```

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 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.

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

So, in this case, it is roughly 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 slightly, 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(gscv2.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](#)

NoteClick [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.misic.metrics import pairwise_distance, lp_distance
```

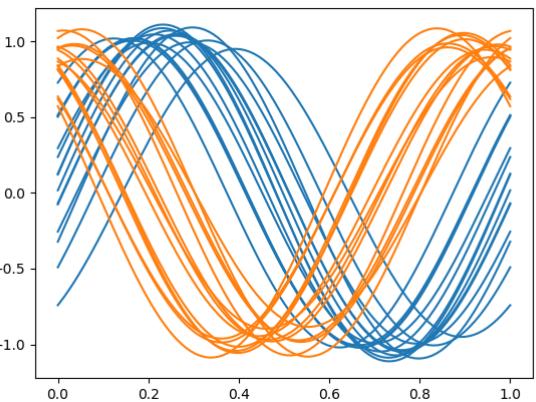
In this example, we are going to show the usage of the radius nearest neighbors classifier in their functional version, a variation of the K-nearest neighbors classifier, where it is used a vote among neighbors within a given radius, instead of use the k nearest neighbors.

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}^∞ 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.

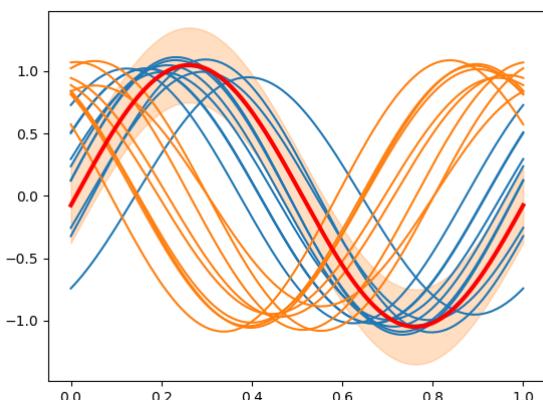
```
plt.figure()

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')
```



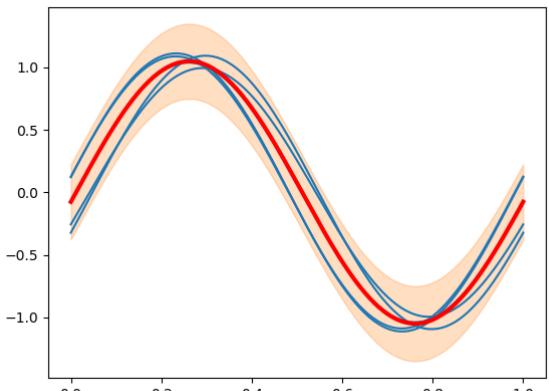
In this case, all the neighbors in the ball belong to the first class, so this will be the class predicted.

```
# 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')
```



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 parameter `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)
```

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

```
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, although, 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 $\|\cdot\|_2$ one, but computationally faster.

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

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

In this dataset $\Delta h = 0.001$, 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 detailed 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 outlier 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]
```

This classifier can be used with multivariate functional data, as surfaces or curves in \mathbb{R}^N , if the metric supports 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.

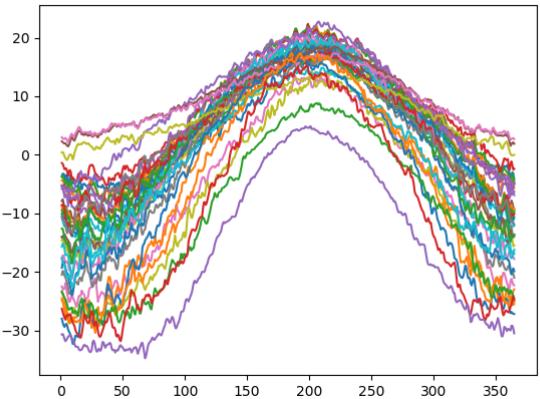
The canadian weather dataset contains the daily temperature and precipitation at 35 different locations in Canada averaged over 1960 to 1994.

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()
```



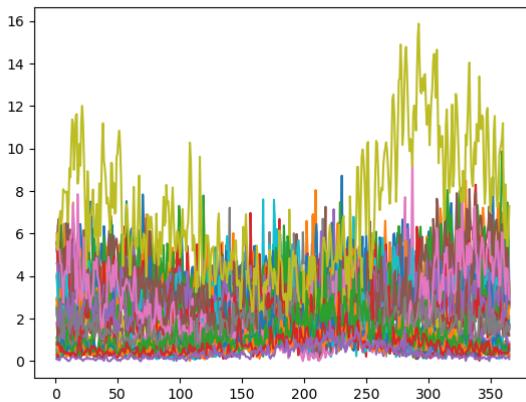
In this example we are not interested in the precipitation curves directly, as in the case with regression response, we will train a nearest neighbor regressor to predict a scalar magnitude.

In the next figure the precipitation curves are shown.

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

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

Canadian Weather



We will try to predict the total log precipitation, i.e., $\log \text{PrecTot}_i = \log \int_0^{365} \text{prec}_i(t) dt$ 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 6.83597705 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 7.0457185 6.73308438 6.40539373 7.85460108 5.58537424
 5.78751069 5.58086126 6.01713229 5.56119432 4.96097889]
```

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

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

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.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")
```

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 York'

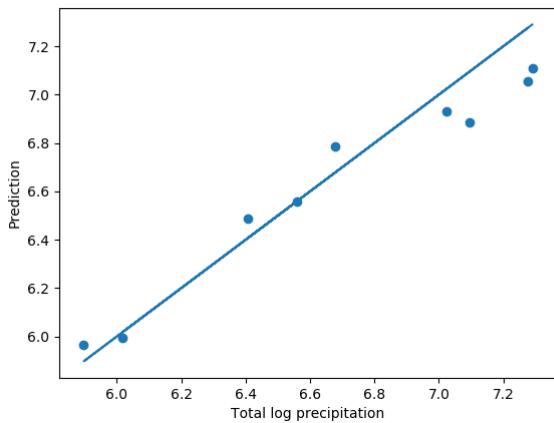
```
plt.show()
```

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



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 $\sum_i(y_i - \hat{y}_i)^2$ 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 approximation 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.

C

PROGRAMMER'S GUIDE

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.

<code>skfda.representation.grid.FDataGrid (data_matrix)</code>	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.

<code>skfda.representation.interpolation.SplineInterpolator ([...])</code>	Spline interpolator of <code>FDataGrid</code> .
--	---

Basis representation

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

<code>skfda.representation.basis.FDataBasis (basis, ...)</code>	Basis representation of functional data.
---	--

The following classes are used to define different basis systems.

<code>skfda.representation.basis.BSpline ([...])</code>	BSpline basis.
<code>skfda.representation.basis.Fourier ([...])</code>	Fourier basis.
<code>skfda.representation.basis.Monomial ([...])</code>	Monomial basis.

Generic representation

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`

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.

<code>skfda.representation.FData (extrapolation, ...)</code>	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`

keepdims

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

Type: `bool`

<code>__init__(extrapolation, dataset_label, axes_labels, keepdims)</code>	[source]
--	----------

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

Methods

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

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

Attributes

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

Parameters:

- `interpolator_order (int, optional)` – Order of the interpolation, 1 for linear interpolation, 2 for quadratic, 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

<code>__init__ ([interpolation_order, ...])</code>	Constructor of the SplineInterpolator.
<code>evaluator (fdatagrid)</code>	Construct a SplineInterpolatorEvaluator used in the evaluation.

Attributes

<code>interpolation_order</code>	Returns the interpolation order
<code>monotone</code>	Returns flag to perform monotone interpolation
<code>smoothness_parameter</code>	Returns the smoothness parameter

[Docs](#) » API Reference » Representation of functional Data » `skfda.representation.interpolation.SplineInterpolator`

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 detailed explanation.

interpolator_order

Order of the interpolation, 1 for linear interpolation, 2 for quadratic, 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.

[Docs](#) » API Reference » Representation of functional Data » Extrapolation

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.

<code>skfda.representation.extrapolation.BoundaryExtrapolation</code>	Extends the domain range using the boundary values.
<code>skfda.representation.extrapolation.ExceptionExtrapolation</code>	Raise and exception.
<code>skfda.representation.extrapolation.FillExtrapolation (...)</code>	Values outside the domain range will be filled.
<code>skfda.representation.extrapolation.PeriodicExtrapolation</code>	Extends the domain range periodically.

Custom Extrapolation

Custom extrapolators could be done subclassing `EvaluatorConstructor`.

<code>skfda.representation.evaluator.EvaluatorConstructor</code>	Constructor of an evaluator.
<code>skfda.representation.evaluator.Evaluator</code>	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__(self, fill_value)

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__(self, fill_value)

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__(self, fill_value)

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,   0.797],
      [ 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__(self, fill_value)

Returns the evaluator used by `FData`.

Returns: Evaluator of the periodic extrapolation.

Return type: (`Evaluator`)

Methods

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

<code>evaluator</code> (fdata)	Construct an evaluator.
--------------------------------	-------------------------

Attributes

<code>fill_value</code>	Returns the fill value of the extrapolation
-------------------------	---

Shift 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.

<code>skfda.preprocessing.registration.shift_registration</code> (fd, *)	Perform shift registration of
<code>skfda.preprocessing.registration.shift_registration_deltas</code> (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](#).

<code>skfda.preprocessing.registration.landmark_shift</code> (fd, ...)	Perform a shift of the curves to
<code>skfda.preprocessing.registration.landmark_shift_deltas</code> (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](#)

<code>skfda.preprocessing.registration.landmark_registration</code> (fd, ...)	Perform landmark registration
<code>skfda.preprocessing.registration.landmark_registration_warping</code> (fd, ...)	Calculate the transform

Elastic Registration

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.

<code>skfda.preprocessing.registration.elastic_registration</code> (...)	Align a Fdatagrid using the
<code>skfda.preprocessing.registration.elastic_registration_warping</code> (...)	Calculate the warping to align

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.

<code>skfda.preprocessing.registration.elastic_mean</code> (...)	Compute the karcher mean under the elastic
<code>skfda.preprocessing.registration.warping_mean</code> (...)	Compute the karcher mean of a set of warpings
<code>skfda.preprocessing.registration.to_srsf</code> (...)	Calculate the square-root slope function (SRSF)
<code>skfda.preprocessing.registration.from_srsf</code> (...)	Given a SRSF calculate the corresponding fud

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 \(2008\)](#).

<code>skfda.preprocessing.registration.mse_decomposition</code> (...)	Compute mean square error measures f
---	--------------------------------------

Utility functions

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

<code>skfda.preprocessing.registration.invert_warping</code> (...)	Compute the inverse of a diffeo
<code>skfda.preprocessing.registration.normalize_warping</code> (warping)	Rescale a warping to normalize it

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.

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 alignment, 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:

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

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].

- **fd** (`FData`) – Functional data object to be registered.
- **maxiter** (`int, optional`) – Maximum number of iterations. Defaults to 5.
- **tol** (`float, optional`) – Tolerance allowable. The process will stop if $\max_i |\delta_i^{(v)} - \delta_i^{(v-1)}| < tol$. 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(...)
```

Docs » API Reference » Preprocessing » Registration »
skfda.preprocessing.registration.shift_registration_deltas

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 alignment, 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:

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

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].

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

- **fd** (`FData`) – Functional data object to be registered.
- **maxiter** (`int, optional`) – Maximum number of iterations. Defaults to 5.
- **tol** (`float, optional`) – Tolerance allowable. The process will stop if $\max_i |\delta_i^{(v)} - \delta_i^{(v-1)}| < tol$. 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.

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

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

Docs » API Reference » Preprocessing » Registration » `skfda.preprocessing.registration.landmark_shift`

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 aligned. If a numeric value is passed the landmarks will be aligned 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 $\frac{1}{2}(\max(\text{landmarks}) + \min(\text{landmarks}))$ which 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

```
>>> 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(...)
```

Docs » API Reference » Preprocessing » Registration » `skfda.preprocessing.registration.landmark_shift_deltas`

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 such 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 aligned. If a numer or list is passed the landmarks will be aligned 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 $\frac{1}{2}(\max(\text{landmarks}) + \min(\text{landmarks}))$ which 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])
```

Docs » API Reference » Preprocessing » Registration » skfda.preprocessing.registration.landmark_registration

skfda.preprocessing.registration.landmark_registration

The registered samples can be obtained with a shift

```
>>> fd.shift(shifts)
FDataGrid(...)
```

```
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 aligned. 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

```
>>> 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()
```

Docs » API Reference » Preprocessing » Registration » skfda.preprocessing.registration.landmark_registration_warping

skfda.preprocessing.registration.landmark_registration_warping

The function will return the registered curves

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

```
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 aligned. 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(...)
```

Docs » API Reference » Preprocessing » Registration » skfda.preprocessing.registration.mse_decomposition

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

$$\text{MSE}_{\text{total}} = \frac{1}{N} \sum_{i=1}^N \int [x_i(t) - \bar{x}(t)]^2 dt$$

We define the constant C_R as

$$C_R = 1 + \frac{\frac{1}{N} \sum_i^N \int [Dh_i(t) - \bar{Dh}(t)][y_i^2(t) - \bar{y}^2(t)] dt}{\frac{1}{N} \sum_i^N \int y_i^2(t) dt}$$

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

$$\begin{aligned} \text{MSE}_{\text{amp}} &= C_R \frac{1}{N} \sum_{i=1}^N \int [y_i(t) - \bar{y}(t)]^2 dt \\ \text{MSE}_{\text{phase}} &= \int [C_R \bar{y}^2(t) - \bar{x}^2(t)] dt \end{aligned}$$

It can be shown that

$$\text{MSE}_{\text{total}} = \text{MSE}_{\text{amp}} + \text{MSE}_{\text{phase}}$$

The squared multiple correlation index of the proportion of the total variation due to phase is defined as:

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

See [KRO8-3] 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 $\text{MSE}_{\text{phase}}$, 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

[KRO8-3] 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)
```

```
>>> f'{mse_amp:.6f}'
'0.000981'
```

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'
```

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.to_srsf

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

Calculate the square-root slope function (SRSF) transform.

Let $f_i : [a, b] \rightarrow \mathbb{R}$ 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, which 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

[SK16-4-1] 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 $f_i : [a, b] \rightarrow \mathbb{R}$ 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, which 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

[SK16-4-2] 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 which will be aligned to the template g . Calculates the warping which minimises the Fisher-Rao distance between g and the registered function $f^*(t) = f(\gamma^*(t)) = f \circ \gamma^*$.

$$\gamma^* = \operatorname{argmin}_{\gamma \in \Gamma} d_\lambda(f \circ \gamma, g)$$

Where d_λ denotes the extended Fisher-Rao distance with a penalty term, used to control the amount of warping.

$$d_\lambda^2(f \circ \gamma, g) = \|SRSF(f \circ \gamma)\sqrt{\gamma} - SRSF(g)\|_{\mathbb{L}^2}^2 + \lambda \mathcal{R}(\gamma)$$

In the implementation it is used as penalty term

$$\mathcal{R}(\gamma) = \|\sqrt{\gamma} - 1\|_{\mathbb{L}^2}^2$$

Which 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, which 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_warping

skfda.preprocessing.registration.elastic_registration_warping(*fdatagrid*,
template=*None*, **lam*=0.0, *eval_points*=*None*, *fdatagrid_srf*=*None*, *template_srf*=*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 which will be aligned to the template g . Calculates the warping which minimises the Fisher-Rao distance between g and the registered function $f^*(t) = f(\gamma^*(t)) = f \circ \gamma^*$.

$$\gamma^* = \operatorname{argmin}_{\gamma \in \Gamma} d_\lambda(f \circ \gamma, g)$$

Where d_λ denotes the extended amplitude distance with a penalty term, used to control the amount of warping.

$$d_\lambda^2(f \circ \gamma, g) = \|SRSF(f \circ \gamma) - SRSF(g)\|_{L^2}^2 + \lambda \mathcal{R}(\gamma)$$

In the implementation it is used as penalty term

$$\mathcal{R}(\gamma) = \|\sqrt{\gamma} - 1\|_{L^2}^2$$

Which 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, which 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_srf** (`FDataGrid`, optional) – SRSF of the fdatagrid, may be passed to avoid repeated calculation.
 - **template_srf** (`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 $\gamma_i : I \rightarrow [a, b]$ be a set of warping functions $\gamma_i : [a, b] \rightarrow [a, b]$ 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. [\[SK16-8-3-2\]](#).

$$\bar{\gamma} = \operatorname{argmin}_{\gamma \in \Gamma} \sum_{i=1}^n d_{FR}^2(\gamma, \gamma_i)$$

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`) – Maximum number of iterations. Defaults to 20.
 - **tol** (`float`) – Convergence criterion, if the norm of the mean of the shooting vectors, $\|\vec{v}\| < tol$, 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

[S11-3-2] 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(*fdatagrid*, *, *lam*=0.0, *center*=*True*,
iter=20, *tol*=0.001, *initial*=*None*, *eval_points*=*None*, *fdatagrid_srf*=*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 be 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

$$[\mu_q] = \operatorname{argmin}_{[q] \in \mathcal{A}} \sum_{i=1}^n d_\lambda^2([q], [q_i])$$

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`) – Maximum number of iterations. Defaults to 20.
 - **tol** (`float`) – Convergence criterion, the algorithm will stop if $|\mu_u^{(nu)} - \mu_u^{(nu-1)}| / 2 \leq tol$.
 - **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_srf** (`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 passed 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- Srivastava, Anuj & Klassen, Eric P. (2016). Functional and shape data analysis. In 8-3-1] *Karcher Mean of Amplitudes* (pp. 273-274). Springer.

[S11- Srivastava, Anuj et. al. Registration of Functional Data Using Fisher-Rao Metric (2011). In 3] *Karcher Mean and Function Alignment* (pp. 7-10). arXiv:1103.3817v2.

Docs » API Reference » Preprocessing » Registration » `skfda.preprocessing.registration.invert_warping`

skfda.preprocessing.registration.invert_warping

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

Compute the inverse of a diffeomorphism.

Let $\gamma : [a, b] \rightarrow [a, b]$ be a function strictly increasing, calculates the corresponding inverse $\gamma^{-1} : [a, b] \rightarrow [a, b]$ 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 $\gamma : [0, 1] \rightarrow [0, 1]$ which 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. ])
```

Docs » API Reference » Preprocessing » Registration » `skfda.preprocessing.registration.normalize_warping`

skfda.preprocessing.registration.normalize_warping

`skfda.preprocessing.registration.normalize_warping(warpings, domain_range=None)`
[source]

Rescale a warping to normalize their domain.

Given a set of warpings $\gamma_i : [a, b] \rightarrow [a, b]$ it is used an affine traslation to change the domain of the transformation to other domain, $\tilde{\gamma}_i : [\tilde{a}, \tilde{b}] \rightarrow [\tilde{a}, \tilde{b}]$.

Parameters:

- `warpings` (`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 L_p 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

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

<code>__init__</code> ([n_neighbors, radius, algorithm, ...])	Initialize the nearest neighbors searcher.
<code>fit</code> (X[, y])	Fit the model using X as training data.
<code>get_params</code> ([deep])	Get parameters for this estimator.
<code>kneighbors</code> ([X, n_neighbors, return_distance])	Finds the K-neighbors of a point.
<code>kneighbors_graph</code> ([X, n_neighbors, mode])	Computes the (weighted) graph of k-Neighbor
<code>radius_neighbors</code> ([X, radius, return_distance])	Finds the neighbors within a given radius of a
<code>radius_neighbors_graph</code> ([X, radius, mode])	Computes the (weighted) graph of Neighbors
<code>set_params</code> (**params)	Set the parameters of this estimator.

```
>>> 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`, `KNeighborsRegressor`, `RadiusNeighborsRegressor`, `NearestCentroids`

Notes

Docs » API Reference » Machine Learning » Classification » 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 L_p 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, 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

<code>__init__</code> ([<code>n_neighbors</code> , <code>weights</code> , <code>algorithm</code> , ...])	Initialize the classifier.
<code>fit</code> (<code>X</code> , <code>y</code>)	Fit the model using <code>X</code> as training data and <code>y</code> as target variable.
<code>get_params</code> ([<code>deep</code>])	Get parameters for this estimator.
<code>kneighbors</code> ([<code>X</code> , <code>n_neighbors</code> , <code>return_distance</code>])	Finds the K-neighbors of a point.
<code>kneighbors_graph</code> ([<code>X</code> , <code>n_neighbors</code> , <code>mode</code>])	Computes the (weighted) graph of k-Neighbors Graphs.
<code>predict</code> (<code>X</code>)	Predict the class labels for the provided data.
<code>predict_proba</code> (<code>X</code>)	Return probability estimates for the test data.
<code>score</code> (<code>X</code> , <code>y</code> , [<code>sample_weight</code>])	Returns the mean accuracy on the given test data and samples.
<code>set_params</code> (** <code>params</code>)	Set the parameters of this estimator.

[Docs](#) » [API Reference](#) » [Machine Learning](#) » [Classification](#) » `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

skfda.ml.classification.RadiusNeighborsClassifier

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 = l_p_distance)`) – The distance metric to use for the tree. The default metric is the L_p 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.

<code>predict (X)</code>	Predict the class labels for the provided data.
<code>radius_neighbors ([X, radius, return_distance])</code>	Finds the neighbors within a given radius of a
<code>radius_neighbors_graph ([X, radius, mode])</code>	Computes the (weighted) graph of Neighbors
<code>score (X, y[, sample_weight])</code>	Returns the mean accuracy on the given test c
<code>set_params (**params)</code>	Set the parameters of this estimator.

```
>>> 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, 1, 1, 1, 1, 1, 1])
```

See also

`KNeighborsClassifier`, `KNeighborsRegressor`, `RadiusNeighborsRegressor`, `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](https://en.wikipedia.org/wiki/K-nearest_neighbour_algorithm)

```
__init__(radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function l_p_distance>, metric_params=None, outlier_label=None, n_jobs=1, sklearn_metric=False) [source]
```

Initialize the classifier.

Methods

<code>__init__ ([radius, weights, algorithm, ...])</code>	Initialize the classifier.
<code>fit (X, y)</code>	Fit the model using X as training data and y as
<code>get_params ([deep])</code>	Get parameters for this estimator.

[Docs](#) » [API Reference](#) » [Machine Learning](#) » [Regression](#) » `skfda.ml.regression.KNeighborsScalarRegressor`

skfda.ml.regression.KNeighborsScalarRegressor

```
class skfda.ml.regression.KNeighborsScalarRegressor(n_neighbors=5, weights='uniform',
algorithm='auto', leaf_size=30, metric=<function l_p_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 = `l_p_distance`)) – The distance metric to use for the tree. The default metric is the L_p 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 l_p_distance>, metric_params=None, n_jobs=1, sklearn_metric=False) [source]
```

Initialize the classifier.

Methods

Docs » API Reference » Machine Learning » Regression » `skfda.ml.regression.RadiusNeighborsScalarRegressor`

skfda.ml.regression.RadiusNeighborsScalarRegressor

```
class skfda.ml.regression.RadiusNeighborsScalarRegressor(radius=1.0, weights='uniform', algorithm='auto', leaf_size=30, metric=<function l_p_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.

<code>__init__</code> ([<code>n_neighbors</code> , <code>weights</code> , <code>algorithm</code> , ...])	Initialize the classifier.
<code>fit</code> (<code>X</code> , <code>y</code>)	Fit the model using <code>X</code> as training data and <code>y</code> as target.
<code>get_params</code> ([<code>deep</code>])	Get parameters for this estimator.
<code>kneighbors</code> ([<code>X</code> , <code>n_neighbors</code> , <code>return_distance</code>])	Finds the K-neighbors of a point.
<code>kneighbors_graph</code> ([<code>X</code> , <code>n_neighbors</code> , <code>mode</code>])	Computes the (weighted) graph of k-Neighbors.
<code>predict</code> (<code>X</code>)	Predict the target for the provided data <code>:para</code>
<code>score</code> (<code>X</code> , <code>y</code> , [<code>sample_weight</code>])	Returns the coefficient of determination R ²
<code>set_params</code> (<code>**params</code>)	Set the parameters of this estimator.

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)
```

<code>score (X, y[, sample_weight])</code>	Returns the coefficient of determination R^2 of
<code>set_params (**params)</code>	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

<code>__init__ ([radius, weights, algorithm, ...])</code>	Initialize the classifier.
<code>fit (X, y)</code>	Fit the model using X as training data and y as target variable.
<code>get_params ([deep])</code>	Get parameters for this estimator.
<code>predict (X)</code>	Predict the target for the provided data :param X :array-like of shape (n_samples, n_features)
<code>radius_neighbors ([X, radius, return_distance])</code>	Finds the neighbors within a given radius of a fixed point.
<code>radius_neighbors_graph ([X, radius, mode])</code>	Computes the (weighted) graph of Neighbors for points in X.

[Docs](#) » [API Reference](#) » [Machine Learning](#) » [Regression](#) » `skfda.ml.regression.RadiusNeighborsFunctionalRegressor`

skfda.ml.regression.RadiusNeighborsFunctionalRegressor

```
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)
```

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 which 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 $5X + 1$.

```
>>> 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_neighbour_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

<code>__init__</code> ([<code>radius, weights, regressor, ...</code>])	Initialize the classifier.
<code>fit</code> (<code>X, y</code>)	Fit the model using <code>X</code> as training data.
<code>get_params</code> ([<code>deep</code>])	Get parameters for this estimator.
<code>predict</code> (<code>X</code>)	Predict functional responses.
<code>radius_neighbors</code> ([<code>X, radius, return_distance</code>])	Finds the neighbors within a given radius of a fdatagrid.
<code>radius_neighbors_graph</code> ([<code>X, radius, mode</code>])	Computes the (weighted) graph of Neighbors for poi-

[Docs](#) » [API Reference](#) » [Machine Learning](#) » [Regression](#) » `skfda.ml.regression.KNeighborsFunctionalRegressor`

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 with 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 which 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

<code>get_params ([deep])</code>	Get parameters for this estimator.
<code>kneighbors ([X, n_neighbors, return_distance])</code>	Finds the K-neighbors of a point.
<code>kneighbors_graph ([X, n_neighbors, mode])</code>	Computes the (weighted) graph of k-Neighbors
<code>predict (X)</code>	Predict functional responses.
<code>score (X, y)</code>	TODO
<code>set_params (**params)</code>	Set the parameters of this estimator.

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_neighbour_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

<code>__init__ ([n_neighbors, weights, regressor, ...])</code>	Initialize the classifier.
<code>fit (X, y)</code>	Fit the model using X as training data.

[Docs](#) » [API Reference](#) » [Machine Learning](#) » [Classification](#) » [skfda.ml.classification.NearestCentroids](#)

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.,
...>>> 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, 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

<code>__init__ ([metric, mean])</code>	Initialize the classifier.
<code>fit (X, y)</code>	Fit the model using X as training data and y as target values.
<code>get_params ([deep])</code>	Get parameters for this estimator.
<code>predict (X)</code>	Predict the class labels for the provided data.
<code>score (X, y[, sample_weight])</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params (**params)</code>	Set the parameters of this estimator.

Docs » API Reference » Datasets » `skfda.datasets.make_sinusoidal_process`

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 process.

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.

Docs » API Reference » Datasets » `skfda.datasets.make_multimodal_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.

$$x_i(t) \propto \sum_{n=1}^{n_{\text{modes}}} \exp\left(-\frac{1}{2\sigma}(t - \mu_n)^T \mathbb{I}(t - \mu_n)\right)$$

Where $\mu_n = \text{mode_location}_n + c$ and c is normally distributed, with mean 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 $\text{points_per_dim}^{\text{ndim_domain}}$ 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.

Docs » API Reference » Datasets » `skfda.datasets.make_multimodal_landmarks`

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 $[0, 1]$

$$v(t) = \sum_{j=0}^N a_j \sin\left(\frac{2\pi j}{K}t\right) + b_j \cos\left(\frac{2\pi j}{K}t\right)$$

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.

