Smooth Functional Principal Component Analysis

Regularization technique for data distributed over planar and two-dimensional manifold domains

Luca Negri

Milano, March 19th, 2018

Abstract

The model here presented is a novel Principal Component Analysis technique suitable for handling functional signals distributed over planar and curved domains(specifically over two-dimensional smooth Riemannian manifolds) that has been proposed and presented by *Lila et al.,2016, Annals of Applied Statistics* ([5]). In this model a regularization approach is used, introducing a smoothing penalty coherent with the geodesic distance. The model can be applied to any planar or manifold topology and can naturally handle missing data or functional samples evaluated in different grids of points. For the discretization task, a finite element approach is used such that the algorithm for its resolution is also very efficient. An efficient `C++` implementation of this model is here presented. The model and the implementation is then tested and its performance is measured and compared to a general MultiVariate PCA. A real application on a Neuroimaging dataset is showed. Finally, the ongoing development of the extension of this model to handle data distributed over volumetric domains is presented.

Contents

1 Introduction

The usage of PDEs in statistics has proved to be particularly useful. In particular, it is possible to use PDEs to incorporate problem-specific prior information about the spatial structure of the phenomenon under study formalized in terms of a governing PDE. This also allows for a very flexible modeling of space variation, that accounts naturally for anisotropy and non-stationarity.

Penalized regression models for the accurate estimation of surfaces and spatial fields, that have regularizing terms involving partial differential operators, have been introduced in [2] and [3].

These models have been extended to handle data distributed over two-dimensional Riemannian manifolds domains, with the construction of a flattening map that maps the manifold into a subspace of $\mathbb{R}^2$; the problem is thus solved over $\mathbb{R}^2$ making use of planar Finite Elements; the solution is then mapped back to the original manifold domain ([4]).

Moreover, based on these regression models, a novel PCA technique suitable for working with functional signals distributed over curved domains has been proposed and studied in [5].

The penalized regression models for the accurate estimation of surfaces and spatial fields has been implemented in the `R` library "fdaPDE", publicly released on CRAN (the Comprehensive R Archive Network)[1]. Some APSC projects have extended this implementation, leading to the creation of multiple versions of the "fdaPDE" library. Specifically, Beraha and Cosmo implemented regression models for data distributed over general two-dimensional Riemannian manifolds, using non planar Finite Elements; this implementation is available at `https://github.com/mariob6/fdaPDE-surface`. Bambini and Giussani implemented a more efficient solution of the estimation problem, via Woodbury decomposition of the matrix involved in the linear system being solved, and the stochastic approximation of the Generalized Cross Validation index for the selection of the smoothing parameter; this implementation is available at `https://github.com/10376920/fdaPDE`. These projects did not target a full compatibility of these alternative "fdaPDE" libraries with the official version available on `CRAN`, so that these features have not been integrated in the official library yet.

The PCA technique proposed in [5] has not yet been implemented in `R/C++`; a `MATLAB` version is available at `https://github.com/eardi/sm-fpca`.

In this report, I present the work that I have been done on the "fdaPDE" library.

The first part of my work consisted in fixing and merging of the different versions of the library. Specifically, a unique version of the "fdaPDE" library has been created, which include all the improvements that has been done in the different projects.

After the creation of the unique version of the "fdaPDE", the library was extended in order to make it able to handle data structures consisting of multiple functional signals instead of only one.

Then, the Smooth Functional Principal Component Analysis algorithm for

data distributed over planar and general two-dimensional Manifold domains has been implemented, extending the library. The implementation of this algorithm has been done in `C++` making it very efficient with respect to the already existing `MATLAB` code.

Further developments of the "fdaPDE" library are being done. The library is being extended to be able to handle functional data distributed over volumetric domains via the implementation of the volumetric Finite Elements. This enables the implementation of regularized regression models and Smooth Functional PCA with volumetric data. These methods represent a major innovation in the field as, currently, there are no techniques for performing regression or PCA models with volumetric data.

2 Mathematical Framework

2.1 Functional Principal Component Analysis

Let $\mathcal{M} \subset \mathbb{R}^3$ be a smooth compact two dimensional Riemannian manifold. We consider the space of the square integrable functions on $\mathcal{M}$: $L^2(\mathcal{M}) = \{f : \mathcal{M} \to \mathbb{R} : \int_{\mathcal{M}} |f(p)|^2 dp < \infty\}$ with the inner product between two functions of this space defined as $\langle f, g \rangle_{\mathcal{M}} = \int_{\mathcal{M}} f(p)g(p)dp$ and the norm defined as $\|f\|_{\mathcal{M}} = \int_{\mathcal{M}} |f(p)|^2 dp$.

We consider also a random variable X with values in $L^2(\mathcal{M})$, mean $\mu = \mathbb{E}[X]$, a finite second moment ($\int_{\mathcal{M}} \mathbb{E}\left[X^2\right] < \infty$) and assume that its covariate function defined as $K(p,q) = \mathbb{E}\left[(X(p) - \mu(p))\,(X(q) - \mu(q))\right]$ is square integrable.

Then, by Mercer's Lemma, the existence of a non increasing sequence (κ_j) of eigenvalues of K and an orthonormal sequence of corresponding eigenfunctions (ψ_j) such that

$$\int_{\mathcal{M}} K(p,q)\psi_j(p)dp = \kappa_j \psi_j(q) \forall q \in \mathcal{M} \tag{1}$$

is guaranteed.

We can rewrite also $K(p,q)$ as $K(p,q) = \sum_{j=1}^{\infty} \kappa_j \psi_j(p)\psi_j(q) \forall p, q \in \mathcal{M}$.

Thus, X can be expanded as $X = \mu + \sum_{j=1}^{\infty} \epsilon_j \psi_j$, where the random variables $\epsilon_1, \epsilon_2, \dots$ are uncorrelated and given by $\epsilon_j = \int_{\mathcal{M}} \{X(p) - \mu(p)\}\psi_j(p)dp$. This is also known as the Karhunen-Love (KL) expansion of X.

The collection $\psi_j(p)$ defines the strongest modes of variation in the random function X and these are called Principal Component (PC) functions. The PC functions are such that

$$\psi_1 = \underset{\phi:\|\phi\|_{\mathcal{M}}=1}{\operatorname{argmax}} \int_{\mathcal{M}} \int_{\mathcal{M}} \phi(p)K(p,q)\phi(q)dpdq$$

and all the subsequent PC function solve the analogous problem with the added constraint of ψ_m being orthogonal to the previous $m-1$ functions $\psi_1 \dots \psi_{m-1}$

$$\psi_m = \underset{\substack{\phi:\|\phi\|_{\mathcal{M}}=1 \\ \langle \phi, \psi_j \rangle_{\mathcal{M}} = 0 \quad \forall j = 1 \dots m-1}}{\operatorname{argmax}} \int_{\mathcal{M}} \int_{\mathcal{M}} \phi(p)K(p,q)\phi(q)dpdq$$

The random variables $\epsilon_1, \epsilon_2, \dots$ are called PC scores.

There are two standard approaches for doing FPCA: the pre-smoothing approach and the regularized PCA approach. We consider the latter, that consists in adding a penalization term to the classic formulation of PCA in order to recover a desired feature of the estimated underlying functions and to encourage the PC functions to be smooth.

Moreover, with this model we are able to handle real functions observable on a two dimensional manifold because we consider a smoothing penalty operator that is coherent with the 2D geodesic distance on the manifold and so we can fully exploit the information about the geometry of the manifold.

2.2 Smooth Functional Principal Component Analysis

Here in this section I will present the Smooth Functional Principal Component Analysis algorithm. The model and the algorithm for its resolution have been proposed and studied in the paper [5]. The model presented in the paper is proposed for data distributed over two-dimensional Riemannian manifolds, but it can be easily extended to handle data distributed over planar domains.

2.2.1 Laplace-Beltrami operator

Before introducing the model, we need to introduce the essential geometric concepts that allow the definition of the Laplace-Beltrami operator. First of all we need to define a base for the tangent space $T_p\mathcal{M}$ of a point $p \in \mathcal{M}$. Let $\varphi : U \subset \mathbb{R}^2 \to \mathbb{R}^3$ be the bijective and smooth function, that is a local parametrization of the point $p \in \mathcal{M}$ and let $\theta \in U$ be such that $\theta = \varphi^{-1}(p)$, then the basis for the tangent space will be

$$\{\frac{\partial \varphi}{\partial \theta_i}(\theta)\}_{i=1,2} \tag{2}$$

We can now equip the Riemannian manifold $\mathcal{M}$ with a metric defining a scalar product g_p on the tangent space $T_p\mathcal{M}$. We can represent g_p as the matrix $G = (g_{ij})_{i,j=1,2}$ such that

$$g_p(v, w) = \sum_{i,j=1}^{2} g_{ij} v_i w_j \tag{3}$$

for all $v = \sum v_i \frac{\partial \varphi}{\partial \theta_i}(\theta)$ and $w = \sum w_i \frac{\partial \varphi}{\partial \theta_i}(\theta)$. We consider then the scalar product induced by the Euclidean embedding space in $\mathbb{R}^3$ and we define

$$g_p ij(\theta) = \frac{\partial \varphi}{\partial \theta_i}(\theta) \cdot \frac{\partial \varphi}{\partial \theta_j}(\theta) \tag{4}$$

We denote also by $G^{-1} = (g^{ij})_{i,j=1,2}$ the inverse and by $g = det(G)$ the determinant of the matrix G. Let now $f : \mathcal{M} \to \mathbb{R}$ be a real valued and twice differentiable function on the manifold $\mathcal{M}$. Let $F = f \circ \varphi$, then the gradient $\nabla_{\mathcal{M}} f$ is defined as

$$(\nabla_{\mathcal{M}} f)(p) = \sum_{i,j=1}^{2} g^{ij}(\theta) \frac{\partial F}{\partial \theta_j}(\theta) \frac{\partial \varphi}{\partial \theta_j}(\theta) \tag{5}$$

The Laplace-Beltrami operator $\Delta_{\mathcal{M}}$ is a generalization of the standard Laplacian defined on $\mathbb{R}^n$ and it is defined as

$$(\Delta_{\mathcal{M}} f)(p) = \frac{1}{\sqrt{g(\theta)}} \sum_{i,j=1}^{2} \frac{\partial}{\partial \theta_j} g^{ij} \sqrt{g(\theta)} \frac{\partial F}{\partial \theta_j}(\theta) \tag{6}$$

2.2.2 Model

We suppose $x_1, \ldots, x_n$ are n smooth samples from a function $X : \mathcal{M} \to \mathbb{R}$; for each of these functions, only noisy evaluations $x_i(p_j)$ on a fixed grid of points $p_1, \ldots, p_s$ in $\mathcal{M}$ are observed. We can now define the $n \times s$ matrix $\mathbf{X} = (x_i(p_j))$. Let now $\mathbf{u} = u_{ii=1\ldots n}$ be a n-dimensional real column vector. The proposed estimate of the first PC function $\hat{f} : \mathcal{M} \to \mathbb{R}$ and the associated PC scores vector $\hat{\mathbf{u}}$ are found by solving the equation

$$(\hat{\mathbf{u}}, \hat{f}) = \underset{\mathbf{u},f}{\operatorname{argmin}} \sum_{i=1}^{n} \sum_{j=1}^{s} (x_i(p_j) - u_i f(p_j))^2 + \lambda \mathbf{u}^T \mathbf{u} \int_{\mathcal{M}} \Delta^2_{\mathcal{M}} f \tag{7}$$

where the Laplace-Beltrami operator is integrated over the manifold $\mathcal{M}$ enabling a global roughness penalty on f, while the empirical term encourages f to capture the strongest mode of variation. The parameter λ controls the trade-off between the empirical term of the objective function and the roughness penalizing term.

The subsequent PCs can be extracted sequentially by removing the preceding estimated components from the data matrix $\mathbf{X}$. In this way we can select a different penalization parameter λ for each PC estimate.

2.2.3 Iterative Algorithm

The algorithm for the resolution of the model and for the minimization of the functional can be splitted in two steps:

- splitting the optimization in a finite dimensional optimization in $\mathbf{u}$ and infinite-dimensional optimization in f;
- Approximating the infinite-dimensional solution thanks to a Surface Finite Element discretization.

We can rewrite the expression (7) as

$$(\hat{\mathbf{u}}, \hat{f}) = \underset{\mathbf{u},f}{\operatorname{argmin}} \|\mathbf{X} - \mathbf{u}\mathbf{f}_s^T\|_F + \lambda \mathbf{u}^T \mathbf{u} \int_{\mathcal{M}} \Delta^2_{\mathcal{M}} f \tag{8}$$

A normalization constraint must be considered in order to make the representation unique. We set this constraint to be $\|\mathbf{u}\|_2 = 1$ so that it allows to leave the infinite-dimensional optimization in f unconstrained. For the minimization of the criterion (8) we alternate the minimization of $\mathbf{u}$ and f in an iterative algorithm:

- *Step 1* Estimation of $\mathbf{u}$ given f. The minimizing $\mathbf{u}$ of the objective function is
$$\mathbf{u} = \frac{\mathbf{X}\mathbf{f}_s}{\|\mathbf{f}_s\|_2^2 + \lambda \int_{\mathcal{M}} \Delta^2_{\mathcal{M}} f} \tag{9}$$
and the unitary-norm vector $\mathbf{u}$ that solve the optimization problem above is
$$\mathbf{u} = \frac{\mathbf{X}\mathbf{f}_s}{\|\mathbf{X}\mathbf{f}_s\|_2} \tag{10}$$

- *Step 2* Estimation of f given $\mathbf{u}$. Finding the minimizing f of the objective function (8) is an equivalent problem to finding the f that minimizes

$$J_{\lambda,\mathbf{u}}(f) = \mathbf{f}_s^T \mathbf{f}_s + \lambda \int_{\mathcal{M}} \Delta_{\mathcal{M}}^2 f - 2\mathbf{f}_s \mathbf{X}^T \mathbf{u} \tag{11}$$

While the problem in *Step 1* is basically the same as the classical expression of finding the score vector given the loadings vector, the problem in *Step 2* is not trivial because it consists in an infinite-dimensional minimization problem. Let z_j denote the j^{th} element of the vector $\mathbf{X}^T\mathbf{u}$, then minimizing (11) is equivalent as minimizing

$$\sum_{j=1}^{s} (z_j - f(p_j))^2 + \lambda \int_{\mathcal{M}} \Delta_{\mathcal{M}}^2 f \tag{12}$$

This problem involves the estimation of a smooth field f defined over a manifold starting from noisy observations z_j at points p_j. In order to solve this problem, a Finite Element discretization approach is needed, where we construct the Finite Element space directly on the triangulated surface $\mathcal{M}_\mathcal{T}$ that approximates the manifold $\mathcal{M}$.

2.2.4 Surface Finite Element discretization

We assume that $\mathcal{M}$ is a closed surface. Consider the linear functional space $H^2(\mathcal{M})$, the space of functions in $L^2(\mathcal{M})$ with first and second weak derivatives in $L^2(\mathcal{M})$. The solution to the infinite dimensional part of the estimation problem can be reformulated in finding $\hat{f} \in H^2(\mathcal{M})$ such that

$$\hat{f} = \underset{f \in H^2(\mathcal{M})}{\operatorname{argmin}} \; J_{\lambda,\mathbf{u}}(f) \tag{13}$$

Proposition 1 *The solution $\hat{f} \in H^2(\mathcal{M})$ exists and is unique and such that*

$$\sum_{j=1}^{s} \varphi(p_j) f(p_j) + \lambda \int_{\mathcal{M}} \Delta_{\mathcal{M}} \varphi \Delta_{\mathcal{M}} \hat{f} = \sum_{j=1}^{s} \varphi(p_j) \sum_{i=1}^{n} x_i(p_j) u_i \tag{14}$$

for every $\varphi \in H^2(\mathcal{M})$.

This is a fourth order problem that cannot be solved analytically. We need to reduce it to a finite dimensional one thanks to a variational reformulation of the problem. We consider now a triangulated surface $\mathcal{M}_\mathcal{T}$ that gives an approximated representation of the manifold $\mathcal{M}$. We introduce also the linear finite element space V consisting in a set of globally continuous functions over $\mathcal{M}_\mathcal{T}$ that are linear affine when restricted to any triangle τ in $\mathcal{T}$, i.e.

$$V = \{v \in C^0(\mathcal{M}_\mathcal{T}) : v|_\tau \; is \; linear \; affine \; for \; each \; \tau \in \mathcal{T}\}.$$

At this point it is possible to construct a lagrangian nodal basis $\psi_1, \ldots, \psi_K$ associated to the nodes $\xi_1, \ldots, \xi_K$ corresponding to the vertices of the triangulation $\mathcal{M}_\mathcal{T}$. Each basis function is such that $\psi_i(\xi_i) = 1$ if $i = j$ and $\psi_i(\xi_i) = 0$ otherwise, i.e. $\psi_i(\xi_j) = \delta_{ij}$. Let $\mathbf{f} = (f(\xi_1), \ldots, f(\xi_K))^T$ be the vector collecting the evaluation of function f at the K nodes and $\boldsymbol{\psi} = (\psi_1, \ldots, \psi_K)^T$ the vector containing the K basis function associated with the K vertices of the triangulation, then every function $f \in V$ can be represented through a basis expansion as:

$$f(p) = \sum_{k=1}^{K} f(\xi_k)\psi_k(p) = \mathbf{f}^T \boldsymbol{\psi}(p) \tag{15}$$

for each $p \in \mathcal{M}_\mathcal{T}$. The surface finite element space provides a finite dimensional subspace of $H^1(\mathcal{M})$ and to use this finite element space to discretize the infinite dimensional problem, that is well posed in $H^2(\mathcal{M})$, we need a reformulation of the infinite dimensional problem. We introduce an auxiliary function g that plays the role of $\Delta_\mathcal{M} f$ and we split the problem (14) into a coupled system of second-order problems.

Proposition 2 *The problem of finding $f \in H^2(\mathcal{M})$ that satisfies (14) for every $\varphi \in H^2(\mathcal{M})$ can be reformulated as the problem of finding $(\hat{f}, g) \in H^2(\mathcal{M}) \times L^2(\mathcal{M})$ that satisfies:*

$$\begin{cases} \sum_{j=1}^{p} \varphi(p_j)\hat{f}(p_j) + \lambda \int_\mathcal{M} (\Delta\varphi) g = \sum_{j=1}^{p} \varphi(p_j) \sum_{i=1}^{n} x_i(p_j) u_i \\ \int_\mathcal{M} vg - \int_\mathcal{M} v(\Delta\hat{f}) = 0 \end{cases} \tag{16}$$

for all $(\varphi, v) \in H^2(\mathcal{M}) \times L^2(\mathcal{M})$.

We get that if the pair of functions $(\hat{f}, g) \in H^2(\mathcal{M}) \times L^2(\mathcal{M})$ satisfies (16) for all $(\varphi, v) \in H^2(\mathcal{M}) \times L^2(\mathcal{M})$, then $\hat{f}$ also satisfies problem (14). In contrast, if $\hat{f} \in H^2(\mathcal{M})$ satisfies problem (14), then the pair $(\hat{f}, \Delta\hat{f})$ automatically satisfies the system of equations (16). It is now possible to integrate by parts the second-order terms to obtain the following:

$$\int_\mathcal{M} (\Delta_\mathcal{M}\varphi) g = -\int_\mathcal{M} \nabla_\mathcal{M}\varphi \nabla_\mathcal{M} g$$
$$\int_\mathcal{M} v(\Delta_\mathcal{M}\hat{f}) = -\int_\mathcal{M} \nabla_\mathcal{M} v \nabla_\mathcal{M}\hat{f}$$

If we also consider the auxiliary function g and the test function v to be such that $g, v \in H^1(\mathcal{M})$, then we can reformulate the problem (14) as finding $(\hat{f}, g) \in (H^1(\mathcal{M}) \cap C^0(\mathcal{M})) \times H^1(\mathcal{M})$:

$$\begin{cases} \sum_{j=1}^{p} \varphi(p_j)\hat{f}(p_j) + \lambda \int_\mathcal{M} \nabla\varphi \nabla g = \sum_{j=1}^{p} \varphi(p_j) \sum_{i=1}^{n} x_i(p_j) u_i \\ \int_\mathcal{M} vg - \int_\mathcal{M} \nabla v \nabla\hat{f} = 0 \end{cases} \tag{17}$$

for all $(\varphi, v) \in (H^1(\mathcal{M}) \cap C^0(\mathcal{M})) \times H^1(\mathcal{M})$. The discrete estimators $(\hat{f}_h, \hat{g}_h) \in V \subset H^1(\mathcal{M})$ can be obtained by solving:

$$\begin{cases} \int_{\mathcal{M}_\mathcal{T}} \nabla_{\mathcal{M}_\mathcal{T}} \hat{f}_h \nabla_{\mathcal{M}_\mathcal{T}} \varphi_h - \int_{\mathcal{M}_\mathcal{T}} \hat{g}_h \varphi_h = 0 \\ \lambda \int_{\mathcal{M}_\mathcal{T}} \nabla_{\mathcal{M}_\mathcal{T}} \hat{g}_h \nabla_{\mathcal{M}_\mathcal{T}} v_h + \sum_{j=1}^{s} \hat{f}_h(p_j) v_h(p_j) = \sum_{j=1}^{s} v_h(p_j) \sum_{i=1}^{n} x_i(p_j) u_i \end{cases} \tag{18}$$

for all $(\varphi_h, v_h) \in V$.

We define the $s \times K$ matrix $\mathbf{\Psi} = (\psi_k(p_j))$ and the $K \times K$ matrices $\mathbf{R}_0 = \int_{\mathcal{M}_\mathcal{T}} (\boldsymbol{\psi}\boldsymbol{\psi}^T)$ and $\mathbf{R}_1 = \int_{\mathcal{M}_\mathcal{T}} (\nabla_{\mathcal{M}_\mathcal{T}} \boldsymbol{\psi})(\nabla_{\mathcal{M}_\mathcal{T}} \boldsymbol{\psi})^T$. We exploit the representation (15) of functions in V and we can rewrite (18) as a linear system. In this way we can characterize the Finite Element solution $\hat{f}_h(p) = \boldsymbol{\psi}(p)^T \hat{\mathbf{f}}$ where $\hat{\mathbf{f}}$ is the solution of

$$\begin{bmatrix} \mathbf{\Psi}^T \mathbf{\Psi} & \lambda \mathbf{R}_1 \\ \lambda \mathbf{R}_1 & -\lambda \mathbf{R}_0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{f}} \\ \hat{\mathbf{g}} \end{bmatrix} = \begin{bmatrix} \mathbf{\Psi}^T \mathbf{X}^T \mathbf{u} \\ \mathbf{0} \end{bmatrix} \tag{19}$$

Solving this linear system leads to

$$\hat{\mathbf{f}} = (\mathbf{\Psi}^T \mathbf{\Psi} + \lambda \mathbf{R}_1 \mathbf{R}_0^{-1} \mathbf{R}_1)^{-1} \mathbf{\Psi}^T \mathbf{X}^T \mathbf{u} \tag{20}$$

Solving the linear system (19) is fast due to the sparsity of the matrix in the left-hand side. For this reason, the algorithm does not find the solution $\hat{\mathbf{f}}$ directly from the formula (20), but it solves the linear system (19) even if it is typically large.

In the model introduced we assumed that all the observed functions x_i are observed and sampled on a common set of points $p_1, \ldots, p_s \in \mathcal{M}$ that coincides with the vertices of the triangulated surface $\mathcal{M}_\mathcal{T}$. However, thanks to the functional nature of the formulation, the model can be easily extended to the case of missing data or sparsely sampled data. Specifically, suppose now that each function x_i is observable on a set of points $p_1^i, \ldots, p_{s_i}^i$, then the natural extension of model (7) becomes:

$$(\hat{\mathbf{u}}, \hat{f}) = \underset{\mathbf{u}, f}{\operatorname{argmin}} \sum_{i=1}^{n} \sum_{j=1}^{s_i} (x_i(p_j^i) - u_i f(p_j^i))^2 + \lambda \mathbf{u}^T \mathbf{u} \int_{\mathcal{M}} \Delta_{\mathcal{M}}^2 f \tag{21}$$

Following the same procedure, we can define an analogous algorithm based on the following two steps:

- *Step 1* Estimation of $\mathbf{u}$ given f. The unitary-norm vector $\mathbf{u}$ minimizing (21) is given by

$$\mathbf{u} \textit{such that } u_i = \frac{\sum_{j=1}^{s_i} x_i(p_j^i) f(p_j^i)}{\sqrt{\sum_{i=1}^{n} (\sum_{j=1}^{s_i} x_i(p_j^i) f(p_j^i))^2}} \tag{22}$$

- *Step 2* Estimation of f given $\mathbf{u}$. The function f minimizing (21) is given by $f = \mathbf{f}^T \boldsymbol{\psi}$ with $\mathbf{f}$ that is the solution of

$$\begin{bmatrix} \mathbf{L} & \lambda \mathbf{R}_1 \\ \lambda \mathbf{R}_1 & -\lambda \mathbf{R}_0 \end{bmatrix} \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} \mathbf{D}^T \mathbf{u} \\ \mathbf{0} \end{bmatrix} \tag{23}$$

where

$$\mathbf{L} = \begin{bmatrix} \sum_{i=1}^{n}\sum_{j=1}^{s_i} u_i^2\psi_1(p_j^i)\psi_1(p_j^i) & \dots & \sum_{i=1}^{n}\sum_{j=1}^{s_i} u_i^2\psi_1(p_j^i)\psi_K(p_j^i) \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{n}\sum_{j=1}^{s_i} u_i^2\psi_K(p_j^i)\psi_1(p_j^i) & \dots & \sum_{i=1}^{n}\sum_{j=1}^{s_i} u_i^2\psi_K(p_j^i)\psi_K(p_j^i) \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} \sum_{j=1}^{s_1} \psi_1(p_j^1)x_1(p_j^1) & \dots & \sum_{j=1}^{s_n} \psi_1(p_j^n)x_n(p_j^n) \\ \vdots & \ddots & \vdots \\ \sum_{j=1}^{s_1} \psi_K(p_j^1)x_1(p_j^1) & \dots & \sum_{j=1}^{s_n} \psi_K(p_j^n)x_n(p_j^n) \end{bmatrix}$$

2.2.5 SM-FPCA algorithm

The algorithm for the resolution of the model SM-FPCA can be summarized in the following steps:

1. Initialization
   (a) Computation of $\mathbf{\Psi}, \mathbf{R}_0, \mathbf{R}_1$
   (b) Perform the SVD: $\mathbf{X} = \mathbf{UDV}^T$
   (c) $\mathbf{f}_s \leftarrow \mathbf{V}[:,1]$, *where* $\mathbf{V}[:,1]$ are the loadings of the first PC
2. Scores estimation:
$$\mathbf{u} \leftarrow \frac{\mathbf{Xf_s}}{\|\mathbf{Xf_s}\|_2}$$
3. PC function's estimation:$\mathbf{f}$ such that
$$\begin{bmatrix} \mathbf{\Psi}^T\mathbf{\Psi} & \lambda\mathbf{R}_1 \\ \lambda\mathbf{R}_1 & -\lambda\mathbf{R}_0 \end{bmatrix} \begin{bmatrix} \mathbf{f} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} \mathbf{\Psi}^T\mathbf{X}^T\mathbf{u} \\ \mathbf{0} \end{bmatrix}$$
4. PC function's evaluation:
$$\mathbf{f}_s \leftarrow \mathbf{\Psi}^T\mathbf{f}$$
5. Repeat Steps 2-4 until convergence
6. Normalization:
$$\hat{f}(p) \leftarrow \frac{\mathbf{f}^T\boldsymbol{\psi}(p)}{\|\mathbf{f}^T\boldsymbol{\psi}\|_{L^2(\mathcal{M}_\mathcal{T})}}$$

Even if the problems (7)-(21) are non-convex minimization problems in $(\mathbf{u}, f)$, the algorithm is efficient because the objective function is non-increasing under its updating rules, since we have the existence and uniqueness of the minimizing f given $\mathbf{u}$ and vice-versa. Moreover, the first guess of the PC function given by the SVD decomposition of the data matrix $\mathbf{X}$ represent a very good starting point and no convergence problems have been detected.

2.2.6 Parameters selection

The proposed model has a smoothing parameter $\lambda > 0$ in order to manage the trade-off between the fidelity of the estimate to the data, via the sum of squared errors, and the smoothness of the solution, via the penalty term. The flexibility given by the smoothing parameter can be seen as an advantageous feature because we can explore data on different scales. But we need to have also a data-driven authomatic method to select the best λ parameter. The selection of the smoothing parameter can be done in two different methods:

- The first approach consists of a K-fold cross validation. The data matrix $\mathbf{X}$ is partitioned by rows into K roughly equal groups. For each of these group of data $k = 1, \ldots, K$ the dataset is split into a validation set $\mathbf{X}^k$, composed of the element of the kth group, and a training set $\mathbf{X}^{-k}$ with the remaining elements. The PC loadings function f^{-k} is estimated on the training set for different values of λ and then the associated score vector $\mathbf{u}^k$ is computed on the validation set. To compute $\mathbf{u}^k$ on the validation set, we use the formula (9) and we approximate the term $\int_{\mathcal{M}} \Delta_{\mathcal{M}}^2$ by $\mathbf{g}^T \mathbf{R}_0 \mathbf{g}$, with $g_h(p) = \boldsymbol{\psi}(p)^T \mathbf{g}$ being the auxiliary function that approximates $\Delta_{\mathcal{M}} f$. The best parameter λ is the one the minimizes the following score:

$$CV(\lambda) = \sum_{k=1}^{K} \frac{\sum_{i=1}^{n} \sum_{j=1}^{s} (x_i(p_j) - u_i^k f^{-k}(p_j))^2}{np} \tag{24}$$

- The second approach is based on the minimization of a generalized cross validation(GCV) criteria integrated on the regression step of the iterative algorithm. Setting $S(\lambda) = \boldsymbol{\Psi}^T (\boldsymbol{\Psi}^T \Psi + \lambda \mathbf{R}_1 \mathbf{R}_0^{-1} \mathbf{R}_1)^{-1} \boldsymbol{\Psi}^T$, the GCV score is then defined as:

$$GCV(\lambda) = \frac{1}{s} \frac{\|(\mathbf{I} - \mathbf{S}(\lambda))(\mathbf{X}^T \mathbf{u})\|^2}{\left(1 - \frac{1}{s} tr\{\mathbf{S}(\lambda)\}\right)^2} \tag{25}$$

  The GCV represents the average misfit of the regression model with a leave-one-out cross validation on the observations' vector $\mathbf{X}^T \mathbf{u}$. Since exluding the ith element from the vector $\mathbf{X}^T \mathbf{u}$ can be considered as removing the ith column from data matrix $\mathbf{X}$, this strategy can be seen as a leave-one-column-out cross validation as opposed to the K-fold approach, where the data matrix $\mathbf{X}$ is partitioned by rows. The GCV approach is generally faster than the K-fold. However, the K-fold does not require the inversion of any matrix and this is an advantageous feature, because it makes this approach applicable and faster in case of datasets $\mathbf{X}$ with a large number of columns s.

2.2.7 Total explained variance

When doing PCA, an important parameter that needs to be chosen and taken into account is the number of PCs that satisfactorily reduces the dimension of

the data. A classical approach for doing that consists in selecting this parameter on the basis of the cumulated explained variance of the PCs. While in the ordinary PCA the scores vector are uncorrelated and their loadings are orthogonal, in this approach neither the loadings are imposed to be orthogonal, nor the scores are uncorrelated. It is nevertheless possible to define and index of explained variance: let $\hat{\mathbf{U}}$ be the $n \times k$ matrix such that the columns of $math\hat{b}fU$ are the first PC scores vector, multiplied by the norm of each associated PC function (since in this approach the scores are normalized to have unitary norm). Without the uncorrelation assumption it is meaningless to compute the total variance explained by the first k PCs by $tr(\hat{\mathbf{U}}^T\hat{\mathbf{U}})$. To overcome this problem, we compute the QR decomposition of $\hat{\mathbf{U}}$ as $\hat{\mathbf{U}} = \mathbf{QR}$ and define the *adjusted total variance* as $\sum_{j=1}^{k} \mathbf{R}_{jj}^2$, where $\mathbf{R}_{jj}^2$ represent the variance explained by the jth PC that is not already explained by the previous $j-1$ components.

2.3 Extension of the model

The model here presented can be easily extended to handle data distributed over planar and/or volumetric domains. The functions and the random variable X have now values in $L^2(D)$, where D is a domain in $\mathbb{R}^2$. The model remains exactly the same, except for the change in the domain. In particular, let $f \in L^2(D)$,with $D \in \mathbb{R}^2$, and u be a real random variable with finite second moment. The first principal component function $\hat{f}$ and its associated first principal component scores vector $\hat{u}$ are found by solving the following equation:

$$(\hat{u}, \hat{f}) = \underset{u,f}{\operatorname{argmin}} \mathbb{E}\left[\int_D \{X - uf\}^2\right] + \lambda \mathbb{E}\left[u^2\right] \int_D \Delta^2 f \tag{26}$$

Here as a roughness penalty term we use the Laplacian, instead of the Laplace-Beltrami operator. The Laplacian operator Δf is defined as $\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$ if we are in a planar domain(i.e. $D \in \mathbb{R}^2$). The Laplacian provides a measure of the local spherical mean of the function f, i.e. the mean value of the function on the surface of a small sphere around the local point taken into account; it is used to stabilize the equation and its role in the penalty term is to compel f to be spatially smooth on the planar or volumetric domain. As in the original model, the subsequent PCs can be extracted sequentially by applying the model to the residuals. The iterative algorithm for the resolution of this model remains the same as before. The only thing that differs is the finite element discretization. The infinite dimensional minimization problem of the two steps algoritm (11) in the planar case becomes:

$$J_{\lambda,\mathbf{u}}(f) = \mathbf{f}_s^T \mathbf{f}_s + \lambda \int_D \Delta^2 f - 2\mathbf{f}_s \mathbf{X}^T \mathbf{u} \tag{27}$$

We assume that D is closed and we assume that $H^2(D)$ is the Hilbert space of all functions which belongs in $L^2(D)$ and with first and second weak derivatives in $L^2(D)$. We need to find $\hat{f} \in H^2(D)$ such that

$$\hat{f} = \underset{f \in H^2(D)}{\operatorname{argmin}} J_{\lambda,\mathbf{u}}(f) \tag{28}$$

Proposition 3 *The solution $\hat{f} \in H^2(D)$ exists and is unique and such that*

$$\sum_{j=1}^{s} \varphi(p_j) f(p_j) + \lambda \int_D \Delta\varphi \Delta\hat{f} = \sum_{j=1}^{s} \varphi(p_j) \sum_{i=1}^{n} x_i(p_j) u_i \tag{29}$$

for every $\varphi \in H^2(D)$.

This, similarly as before, is a fourth order problem that cannot be solved analytically. We need to reduce it to a finite dimensional one thanks to a variational reformulation of the problem. We consider now a triangulated surface $D_{\mathcal{T}}$ that gives an approximated representation of the planar domain D. We introduce also the linear finite element space V consisting in a set of globally continuous functions over $D_{\mathcal{T}}$ that are linear affine where restricted to any triangle τ in $\mathcal{T}$, i.e.

$$V = \{v \in C^0(D_{\mathcal{T}}) : v|_\tau \textit{ is linear affine for each } \tau \in \mathcal{T}\}.$$

At this point it is possible to construct a lagrangian nodal basis $\psi_1, \ldots, \psi_K$ associated to the nodes $\xi_1, \ldots, \xi_K$ corresponding to the vertices of the triangulation $D_{\mathcal{T}}$. $\psi_i(\xi_j) = \delta_{ij}$. Let $\mathbf{f} = (f(\xi_1), \ldots, f(\xi_K))^T$ be the vector collecting the evaluation of function f at the K nodes and $\boldsymbol{\psi} = (\psi_1, \ldots, \psi_K)^T$ the vector containing the K basis function associated with the K vertices of the triangulation, then every function $f \in V$ can be represented through a basis expansion as:

$$f(p) = \sum_{k=1}^{K} f(\xi_k)\psi_k(p) = \mathbf{f}^T \boldsymbol{\psi}(p) \tag{30}$$

for each $p \in D_{\mathcal{T}}$. The planar finite element space provides a finite dimensional subspace of $H^1(D)$ and to use this finite element space to discretize the infinite dimensional problem, that is well posed in $H^2(D)$, we need a reformulation of the infinite dimensional problem. We introduce an auxiliary function g that plays the role of Δf in order to split the problem (29) into a coupled system of second-order problems. The problem of equation (29) is reformulated as finding $(\hat{f}, g) \in (H^1(D) \cap C^0(D)) \times H^1(D)$:

$$\begin{cases} \sum_{j=1}^{p} \varphi(p_j)\hat{f}(p_j) + \lambda \int_D \nabla\varphi\nabla g = \sum_{j=1}^{p} \varphi(p_j) \sum_{i=1}^{n} x_i(p_j) u_i \\ \int_D vg - \int_D \nabla v \nabla \hat{f} = 0 \end{cases} \tag{31}$$

for all $(\varphi, v) \in (H^1(D) \cap C^0(D)) \times H^1(D)$. $\hat{f}$ is still guaranteed to belong in $H^2(D)$.

The discrete estimators $(\hat{f}_h, \hat{g}_h) \in V \subset H^1(D)$ are then obtained by solving 30:

$$\begin{cases} \int_{D_{\mathcal{T}}} \nabla \hat{f}_h \nabla \varphi_h - \int_{D_{\mathcal{T}}} \hat{g}_h \varphi_h = 0 \\ \lambda \int_{D_{\mathcal{T}}} \nabla \hat{g}_h \nabla v_h + \sum_{j=1}^{s} \hat{f}_h(p_j) v_h(p_j) = \sum_{j=1}^{s} v_h(p_j) \sum_{i=1}^{n} x_i(p_j) u_i \end{cases} \tag{32}$$

for all $(\varphi_h, v_h) \in V$. We can rewrite (32) as a linear system and solve it similarly as in (19).

3 Code Structure

The implementation steps that I followed for extending the `R` library "fdaPDE" and making it able to perform the Smooth Functional Principal Component Analysis on manifold and planar domains were the following:

- the fix and the merge of the existing code of the different versions of the "fdaPDE" library;
- a class for handling multiple data signals and the `R/C++` interface;
- a Factory for the selection of the different cross-validation method;
- a class for the implementation of the SF-PCA algorithm;

3.1 Fix and merge of "fdaPDE" libraries

Before starting with the implementation of the Smooth Functional Principal Component Analysis, I needed to make some adjustments and fixes to the already existing code of the "fdaPDE" `R` library. As already said in the introducion, due to the different APSC projects that have been done on this library, there were different versions of the library:

- the original version of the code done by E.Lila and available at `https://github.com/eardi/fdaPDE`
- the extension of the library able to handle data distributed over manifold domains, an APSC project done by M.Beraha and A.Cosmo, with the code available at `https://github.com/mariob6/fdaPDE-surface`
- some modifications and improvements about the solution of the linear system of the Regression Model and the computation of the GCV, an APSC project done by C.Bambini and L.Giussani, with the code available at `https://github.com/10376920/fdaPDE`

These three versions of the library were developed separately: the APSC projects were based on a previous version of Lila's code and they were also incompatible between each other. The first part of the work consisted in the merging of the different versions and the creation of a unique version with all the improvements that have been done separately.

The base code that I used as a starting point was the updated version of Lila's code.

The first step was to add the possibility to handle data distributed over manifold domains and to perform linear models with differential regularization on non-planar manifold domains. In order to do so, some changes had to be done to the original version of Lila's code, both in `R` and in `C++`.

In `R`, the function `smooth.FEM.basis` was modified in order to make it become a wrapper function for performing regression models over both planar and

manifold domains. The `smooth.FEM.basis` will call the `CPP_smooth.FEM.basis` or the `CPP_smooth.manifold.FEM.basis` according to the type of the object representing the mesh(`MESH2D` or `MESH.2.5D` respectively) that it receives as an input. The `smooth.FEM.basis` is the only function that will be called by the user for performing regression models, regardless of the type of the mesh object.

Following the same logic, also the `plot.FEM` function was modified in order to be able to handle different mesh types. This function will call one of the functions for doing the plot on a planar domain according to the order(`R_plot.ORD1.FEM` or `R_plot.ORDN.FEM`) or the function for doing the plot on a manifold domain(`R_plot_manifold`).

Listing 1: `R` function for plotting functional values

```
plot.FEM = function(x, num_refinements = NULL, ...) {
    if(class(x$FEMbasis$mesh)=="MESH2D"){
        if(x$FEMbasis$order==1){
            R_plot.ORD1.FEM(x,...)
        }else{
            R_plot.ORDN.FEM(x,num_refinements, ...)
        }
    }else if(class(x$FEMbasis$mesh)=="MESH.2.5D"){
        R_plot_manifold(x,...)
    }
}
```

The `R` function for plotting a function on a manifold domanin, `R_plot_manifold`, was also modified in order to make it more efficient than its previous implementation and in order to make it consistent with the functions that plot functional data on a planar domain. The `rgl` library was used and, in particular, the triangular elements of the mesh are being plotted using the command `rgl.triangle`.

In `C++`, the function `SEXP regression_Laplace` was modified in order to make it the unique function for performing the linear models over both planar and manifold domains. This function is responsible for setting the template parameters `ndim`(that specifies if the nodes of the mesh lies in $\mathbb{R}^2$ or $\mathbb{R}^3$) and `mydim`(that quantifies the "local dimension" of the object considered).

Following the work of M.Beraha and A.Cosmo, these template paremeters and relative template specialization has been added to most of the classes of the `C++` code. These classes are the ones responsible for handling the mesh, the mesh elements and computing the Finite Element.

Listing 2: Example: the class MeshHandler

```
template <UInt ORDER,UInt mydim, UInt ndim>
class MeshHandler{
};
```

In the constructor of the class `MeshHandler`, an indexing problem was found for the vector containing the coordinates of the nodes of the mesh and for the vector of the elements. This problem was generated by the different indexing between `R` and `C++` and a fix to this problem was done.

The next step was to modify the class responsible for the resolution of the linear system and change the methods that compute the GCV, adding the choice between an exact and a stochastic computation of it, following the work of C.Bambini and L.Giussani.

In `R`, in the `smooth.FEM.basis` function, two parameters were added: `GCVmethod`, a parameter that let the user specify the type of the GCV computation that he wants to perform, and `nrealizations`, that specifies the number of realizations to do for the stochastic computation of the GCV. These parameters are then passed among the others to `C++` and are handled in the class `RegressionData`, a `C++` class responsible for converting the `R` objects into `C++` objects.

The class responsible for the resolution of the linear system is the class `mixedFERegression`. New methods that are needed for the Woodbury decomposition of the linear system were added: `system_factorize`, a method for the factorization of a matrix, `system_solve`, a method for an efficient resolution of the linear system taking advantage of the Woodbury decomposition, and `LeftMultiplybyQ`, a method that efficiently implements the multiplication of a matrix times a vector. Also for the computation of the GCV, new methods were added to the class `mixedFERegression`: `computeDegreesOfFreedom` in which there is a switch, based on the value of the `GCVmethod` parameter previously introduced, that is responsible for the choice between the exact and the stochastic computation of the GCV; `computeDegreesOfFreedomExact` that is an efficient implementation for the computation of the GCV based on the Woodbury decomposition; `computeDegreesOfFreedomStochastic` that is the implementation of the stochastic method for computing the GCV.

These methods were introduced in the APSC project of C.Bambini and L.Giussani and has been adapted to the new version of the "fdaPDE" library.

With these modifications, the latest version of Lila's code was improved and a unique and better version of "fdaPDE" library was created. After the unification of all the existing libraries, I could start with the implementation of the Smooth Functional Principal Compinent Analysis.

3.2 `R/C++` interface: the class `FPCAdata`

In order to let `R` comunicate with the `C++` code and to pass the data structures we need a particular interface, that is described in [7]. To be coherent with the previous code, the interface used is the one named `.Call`.

On the `R` side, you can call a function passing general `R` objects by:

Listing 3: Usage of .Call in R

```
dyn.load ("fdaPDE.so")
```

```
.Call("Smooth_FPCA",locations, datamatrix, ...,
package = "fdaPDE")
```

On C++ side, we will have something like this code:

Listing 4: C++ function to call in R

```
#include <R.h>
#include <Rinternals.h>
extern "C" {
SEXP Smooth_FPCA(SEXP Rlocations,SEXP Rdatamatrix,...){
    ...
    SEXP result = PROTECT(Rf_allocVector(VECSXP, 7));
    ...
    UNPROTECT(1);

    return result;
}
}
```

SEXP stands for "S Expression" and it is a pointer to a C struct able to handle all types of R objects, that can be of different types: REALSXP(numeric), INTSXP(integer),LGLSXP(logical), VECSXP(list), CHARSXP(character) among others.

You can create also R object in C, but you need to tell R about the object by wrapping the creation in PROTECT() in order to prevent R from garbage collecting it. The command UNPROTECT() simply unprotects the last n objects protected.

The C++ file needs to be compiled with the command R CMD SHLIB in order to create a shared object that can be dynamically loaded in R with the command dyn.load("fdaPDE.so").

In the *src* directory of the project we need a *Makevars* file containing all the flags for compiling and linking.

Using this interface, the multiple data signals are passed from R to C++. To handle these data structures, the class FPCAdata is created.

Listing 5: FPCAdata class

```
class  FPCAData{
private:
    std::vector<Point> locations_;
    bool locations_by_nodes_;
    MatrixXr datamatrix_;
    std::vector<UInt> observations_indices_;
    UInt order_;
    std::vector<Real> lambda_;
    UInt nPC_;
```

```
    UInt nFolds_;
    ...
}
```

This class is responsible for the creation and the initialization of the C++ objects needed for the SF-PCA algorithm. It stores information about the locations of the observations if they don't coincide with the nodes of the mesh, the datamatrix containing multiple functional data signals, the vector containing all the possible values of the smoothing parameter λ, the number of PCs to compute and the number of folds to use if the chosen cross-validation method is the K-fold.

To build the C++ objects, "special" constructors are needed for converting the SEXP objects into standard C++ types.

Listing 6: FPCAdata constructor

```
#ifdef R_VERSION_
FPCAData::FPCAData(SEXP Rlocations, SEXP Rdatamatrix,
SEXP Rorder, SEXP Rlambda, SEXP RnPC, SEXP RnFolds,
SEXP RGCVmethod, SEXP Rnrealizations){
    setLocations(Rlocations);
    setDatamatrix(Rdatamatrix);
    order_=INTEGER(Rorder)[0];
    UInt length_lambda = Rf_length(Rlambda);
    for (UInt i = 0; i<length_lambda; ++i)
        lambda_.push_back(REAL(Rlambda)[i]);
    nPC_ = INTEGER(RnPC)[0];
    nFolds_=INTEGER(RnFolds)[0];
}
```

The datamatrix that handle multiple data signals is built in the method setDatamatrix

Listing 7: setDatamatrix method

```
void FPCAData::setDatamatrix(SEXP Rdatamatrix)
{
    n_=INTEGER(Rf_getAttrib(Rdatamatrix,R_DimSymbol))[0];
    p_=INTEGER(Rf_getAttrib(Rdatamatrix,R_DimSymbol))[1];
    datamatrix_.resize(n_,p_);
    observations_indices_.reserve(p_);
    VectorXr auxiliary_row_;
    auxiliary_row_.resize(p_);
    if(locations_.size() == 0)
    {
        locations_by_nodes_ = true;
        for(auto i=0; i<n_; ++i)
        {
```

```
            UInt count=0;
            for(auto j=0; j<p_ ; ++j)
            {
                if(!ISNA(REAL(Rdatamatrix)[i+n_*j]))
                {
                    auxiliary_row_[count]=REAL(Rdatamatrix)[i+n_*j];
                    count++;
                    if(i==0) observations_indices_.push_back(j);
                }
            }
            datamatrix_.row(i)=auxiliary_row_;
        }
        datamatrix_.conservativeResize(Eigen::NoChange,
        observations_indices_.size());
    } else {
        locations_by_nodes_ = false;
        for(auto i=0; i<n_; ++i)
        {
            for(auto j=0; j<p_ ; ++j)
            {
                datamatrix_(i,j)=REAL(Rdatamatrix)[i+n_*j];
            }
        }
    }

}
```

With this class, the problem of passing data structures from `R` to `C++` is solved and we can proceed with the implementation of the algorithm.

3.3 Implementation of the algorithm: a Factory for the selection of the cross-validation method

The implementation of the SF-PCA algorithm has been done in the class `MixedFEFPCABase` and in all of its children.

A very important part of the algorithm is the selection of the best smoothing parameter λ and we need a tool to automatically select it according to some criteria. The proposed criteria are two: the Generalized Cross Validation(GCV) and the K-fold cross validation. Other than that, we need also a version of the algorithm in which we do not need to select the best λ because we have only one value of it. The algorithm implementation is slightly different according to the different type of cross validation method chosen. The different version of the algorithm has been implemented in the classes `MixedFEFPCA`, `MixedFEFPCAGCV` and `MixedFEFPCAKFold` .The selection of the cross validation method has to be done at runtime, based on some parameters that the user passes to the `R` function `smoooth.FEM.FPCA`. This parameter is a string parameter, `validation`, that by

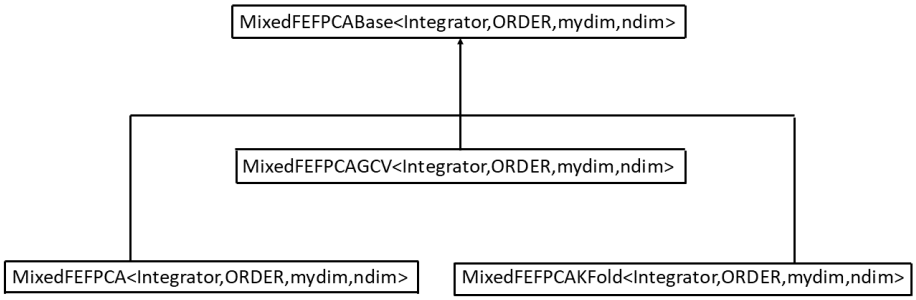

Figure 1: Inheritance graph for class `MixedFEFPCABase`

default is `NULL`, but has to be specified as `"GCV"` or `"KFold"` if the dimension of the parameter λ is greater than one, otherwise `validation` will assume the value `"NoValidation"`.

In `C++`, this parameter is than used in a Factory Method design pattern, defined in the `C++` file "mixedFEFPCAfactory.h", to create the right object: the Factory returns a unique pointer to an object of a derived class of `MixedFEFPCABase` according to the different values of the string parameter `validation`

Listing 8: mixedFEFPCAfactory class

```
template<typename Integrator, UInt ORDER, UInt mydim, UInt ndim>
class MixedFEFPCAfactory
{
    public:
    static std::unique_ptr<MixedFEFPCABase<Integrator, ORDER,
mydim,  ndim>> createFPCAsolver(const std::string &validation,
    const MeshHandler<ORDER,mydim,ndim>& mesh, const FPCAData& fpcaData)
    {
        if(validation=="GCV") return make_unique<MixedFEFPCAGCV
        <Integrator,ORDER, mydim, ndim>>(mesh,fpcaData);
        if(validation=="KFold") return make_unique<MixedFEFPCAKFold
        <Integrator, ORDER, mydim, ndim>>(mesh,fpcaData);
        if(validation=="NoValidation") return make_unique<MixedFEFPCA
        <Integrator, ORDER, mydim, ndim>>(mesh,fpcaData);
    }
};
```

`MixedFEFPCABase` is a particular virtual class with only the method `apply` being virtual and all the other methods that are instead non-virtual because they are methods common to all the different versions of the algorithm. In the derived classes of `MixedFEFPCABase` class, the method `apply` is specified accordingly and some other specific methods are added.

With this programming pattern, in the main file of the library "fdaPDE.cpp",

in order to call the method that perform the SF-PCA algorithm, it is sufficient to create the object using the Factory and then call the virtual method apply:

Listing 9: Usage of MixedFEFPCAfactory and `apply` method in "fdaPDE.cpp"

```
std::unique_ptr<MixedFEFPCABase<Integrator, ORDER, mydim, ndim>>
fpca=MixedFEFPCAfactory<Integrator, ORDER, mydim, ndim>::
 createFPCAsolver(validation, mesh,fPCAData);

fpca->apply();
```

With the use of this programming pattern, it will be also very simple to extend the code, if needed, and add a new cross validation method in the future.

3.4 Implementation of the algorithm: the class `MixedFEFPCABase` and its children

As stated in the previuos subsection, the algorithm of the Smooth Functional Principal Component Analysis is implemented in the class `MixedFEFPCABase` and in all of its children classes. In the `MixedFEFPCABase` class are defined all the methods common to all the child classes: these are the methods responsible for building all the single parts of the Linear System of equation (19). These are the methods:

- `computeBasisEvaluations`, that compute the $\mathbf{\Psi}$ matrix;
- `computeDataMatrix` and `computeDataMatrixByIndices`, that compute the North-West block of the linear system;
- `computeRightHandData`, that computes the right hand side of the system;
- `buildCoeffMatrix`, that puts together all different blocks of the matrix of the system;

Apart from those methods, there are other methods that are common to all the children of `MixedFEFPCABase` class and these are:

- `SetAndFixParameters`, a method responsible for calling the `Assembler` class in order to build the Mass and Stiffness matrices needed for the system and also fix the correct size of all the objects;
- `computeIterations`, a method that computes steps 2.,3.,4.and 5. of the algorithm explained at subsection 2.2.5 for a fixed parameter λ;
- `computeVarianceExplained` and `computeCumulativePercentageExplained`, methods that compute the explained variance of the first k PC found by the algorithm, as explained in the subsection 2.2.7;

In the `computeIterations` method, the estimated loadings, the estimated scores and the observation data needed for the right hand side part of the system ($\mathbf{\Psi}^T\mathbf{X}^T\mathbf{u}$) are at first initialized and then updated at every iteration in the object of class `FPCAObject`.

Listing 10: `FPCAObject class`

```
class  FPCAObject{
private:

    VectorXr scores_;
    VectorXr loadings_;

    VectorXr ObservationData_;
    ...
}
```

For the initialization, the header version of the library "RedSVD"(`https://github.com/ntessore/redsvd-h`) is used for computing the SVD of the data matrix. The particularity of the SVD decomposition of this library is that it is a truncated SVD and so, you can specify the number of singular values (and corresponding vectors) that you want to compute. Since for the algorithm described in 2.2.5 we only need the vector associated to the first singular value of the matrix as a starting point for the iterations, the choice of this library seems very appropriate because it makes the computation of the first loadings vector very fast, even for large data matrices.

In the derived classes of `mixedFEFPCABase`, the apply method is specified and other methods are added if needed:

- in the class `mixedFEFPCA`, the apply method simply calls the methods defined in the father class and the implementation of the algorithm is very straight-forward;
- in the class `mixedFEFPCAGCV`, there are some methods for the computation of the degrees of freedom and the computation of GCV. The Exact computation is done with the usage of the `MUMPS` library in case of locations of the observated data coinciding with the nodes of the mesh, or with `Eigen` in the other case.

  The `apply` method authomatically performs the selection of the best parameter λ for every PC computed using the methods for the GCV computation;
- in the class `mixedFEFPCAKFold`, a method called `computeKFolds` is responsible for dividing the data matrix in training and validation set and to evaluate the CV score for each value of λ.

  In the `apply` method, the best λ is selected and the algorithm 2.2.5 is applied to the entire data matrix with the chosen λ, for each PC that needs to be computed;

The result of these algorithm are then passed to the main function in the `C++` file "fdaPDE.cpp", that is responsible for creating the `R` object and pass it to `R` as explained in subsection 3.2.

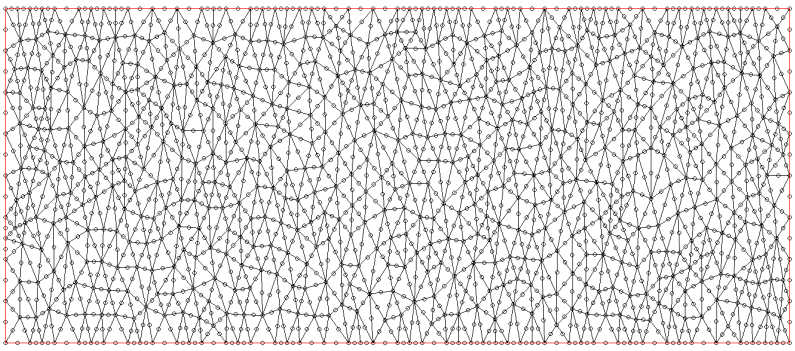

Figure 2: Mesh of a rectangle with 2465 nodes

4 Results

After the implementation of the code, the next important thing to do is to check the performance of it, both in terms of accuracy and computational cost. The Smooth Manifold Functional Principal Analysis model has also been implemented by E.Lila in a `MATLAB` version (the code is available at `https://github.com/eardi/sm-fpca`), so the comparison in both accuracy and performance has been done with that version of the code. In this implementation, only a K-fold cross-validation approach for selecting the best parameter λ is present. Regarding the accuracy, a comparison has been done also with a simple standard Multi Variate-PCA(MV-PCA) applied directly to the data matrix $\mathbf{X}$. This method does not take into account any information about the geometry of the object and so the PC functions with this method are obtained by piecewise linear interpolation over the mesh.

The tests were performed on a machine with a `Intel`® `Core`™ `i7-6700HQ Quad Core CPU @2.60 GHz`, with `16 Gb RAM`.

4.1 Simulation on a planar domain

The proposed Smooth Functional Principal Component Analysis algorithm has been implemented and works very well in planar domains, domains that fully belong to $\mathbb{R}^2$. By specifying the template parameters `mydim` and `ndim`, the method `apply` of the object `mixedFEFPCABase` and of its children has the flexibility of being suitable for both planar and manifold domains. Here in this section I will present a test of this algorithm on a simple planar domain. I considered a simple triangular mesh representing a rectangle, with 2465 nodes(in Figure 2)

In order to run the simulation, I generated $n = 100$ test smooth functions with values on $\mathbb{R}^2$. The smooth functions were generated using the formula:

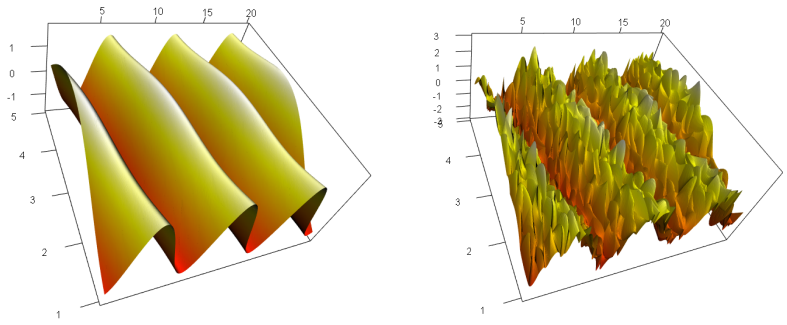

Figure 3: Left: example of a smooth function x_i generated according to the formula (33). Right: same smooth function x_i plus noisy term. In these plots, the domain is the planar rectangle and both the third axes and the color give a representation of the value of the function.

$$x_i = a_{i1} f_1 + a_{i2} f_2 \qquad i = 1, \ldots, n \tag{33}$$

where $f1$ and $f2$ represent the two PC functions with expression:

$$\begin{cases} f_1(x, y) = sin(x) * cos(y) \\ f_2(x, y) = cos(x) * sin(x) \end{cases} \tag{34}$$

and a_{i1},a_{i2} represent the PC scores generated indipendently and distributed as $a_{i1} \sim N(0, \sigma_1^2)$, $a_{i2} \sim N(0, \sigma_2^2)$ with $\sigma_1 = 2$ and $\sigma_2 = 1$. The PC functions are sampled at the locations corresponding to the nodes of the mesh. At these locations, a Gaussian white noise with standard deviation $\sigma = 0.5$ has been added to the true function x_i. We are interested in recovering the smooth PC function f_1 and f_2 from these noisy observations. An example of an observation x_i and the corresponding observation x_i plus the noisy term is showed in Figure

The representation of the original two PC functions on the rectangular planar domain is shown in Figure 4.

The recovered principal component functions using the Smooth Functional Principal Component Analysis algorithm to the dataset, with the selection of the best parameter λ using the K-fold method, are represented in Figure 5, while the recovered principal components functions using a MultiVariate PCA are showed in Figure6

The algorithm, starting from a dataset containing 100 samples of the smooth function x_i plus noise, is able to fully recover and distinguish the two Principal Component functions. The Principal Component functions reconstructed using MultiVariate PCA, instead, are not very smooth and this is more evident in the reconstruction of the second Principal Component function.

To assess the performance of the two methods, I repeated this test 100 times and computed the MSE between the original Principal Component functions

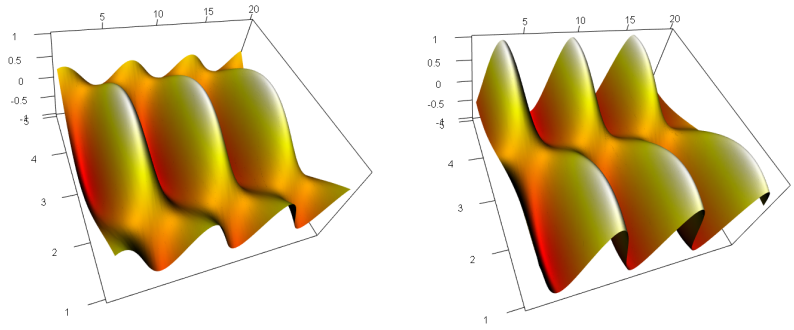

Figure 4: Left: representation of the first principal function f_1. Right: representation of the second principal function f_2

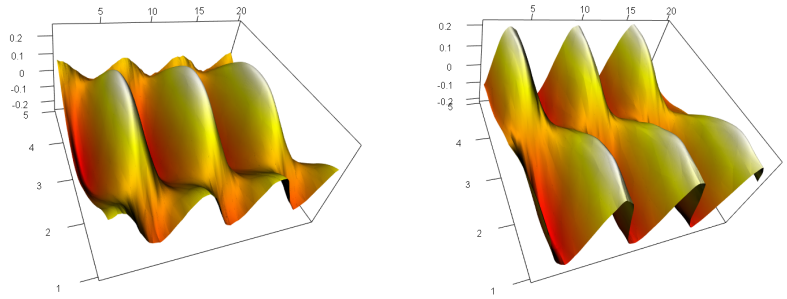

Figure 5: Left: recovered first principal function f_1 using the Smooth Functional Principal Component Analysis with the K-fold cross validation to select the parameter λ. Right: recovered second principal function f_2 using the Smooth Functional Principal Component Analysis with the K-fold cross validation to select the parameter λ.

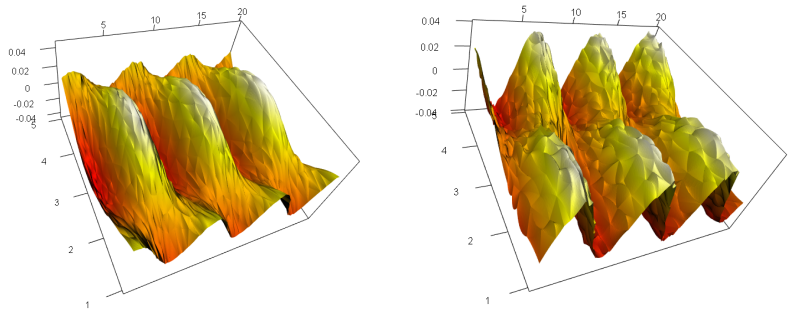

Figure 6: Left: recovered first principal function f_1 using MV-PCA. Right: recovered second principal function f_2 using MV-PCA

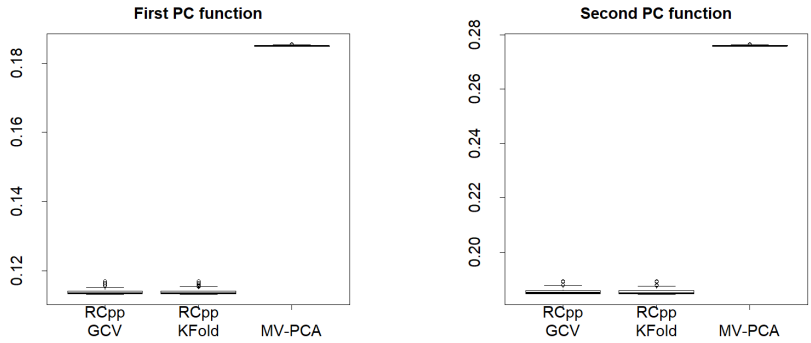

Figure 7: Boxplots summarizing the accuracy performance of MV-PCA and Smooth Functional PCA with a function on a planar domain

and the recovered ones. For the SF-PCA algorithm the selection of the best parameter λ has been done using both K-fold cross validation and GCV. The results are showed in Figure 7.

The reconstruction of the Principal Component Functions, as we can see, is much more faithful if we use the Smooth Functional Principal Component Analysis algorithm, as the error is much smaller indipendently from the cross-validation method used for the selection of λ.

4.2 Simulation on the sphere - Accuracy

In this section I will illustrate the result of the proposed algorithm on a triangulated surface $\mathcal{M}_{\mathcal{T}}$ that is an approximation of a surface of a sphere centered in the origin and with radius $r = 1$. In this test, I will use a mesh representing the sphere, with number of nodes equal to 1982:

In order to run the simulation, I generated $n = 1000$ test smooth functions

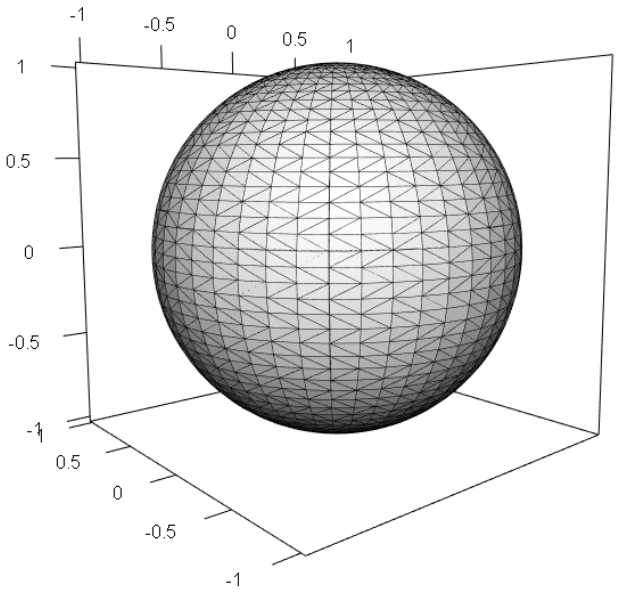

Figure 8: Mesh of a sphere with 1982 nodes

$x_1, \ldots, x_{1000}$ with values on $\mathcal{M}_{\mathcal{T}}$ by:

$$x_i = u_{i1}v_1 + u_{i2}v_2 \qquad i = 1, \ldots, n \tag{35}$$

where $v1$ and $v2$ represent the two PC functions with expression:

$$\begin{cases} v_1(x, y, z) = \frac{1}{2}\sqrt{\frac{15}{\pi}\frac{xy}{r^2}} \\ v_2(x, y, z) = \frac{3}{4}\sqrt{\frac{35}{\pi}\frac{xy(x^2-y^2)}{r^4}} \end{cases} \tag{36}$$

and u_{i1},u_{i2} represent the PC scores generated indipendently and distributed as $u_{i1} \sim N(0, \sigma_1^2)$, $u_{i2} \sim N(0, \sigma_2^2)$ with $\sigma_1 = 4$ and $\sigma_2 = 2$. The PC functions are sampled at the locations corresponding to the nodes of the mesh. At these locations, a Gaussian white noise with standard deviation $\sigma = 0.1$ has been added to the true function x_i. We are interested in recovering the smooth PC function v_1 and v_2 from these noisy observations. The representation of the original two PC functions on the surface of the sphere is showed in Figure 9

I applied the proposed Smooth Functional Principal Component Analysis algorithm in the `R/C++` implementation, choosing the optimal smoothing parameter λ with both the K-fold and GCV validation methods, and I compared it with the `MATLAB` implementation(with only the K-fold cross validation method) and a general MultiVariate PCA applied on the datamatrix.

For assessing the performance, I computed the MSE between the original principal component functions and scores and the ones found by the different

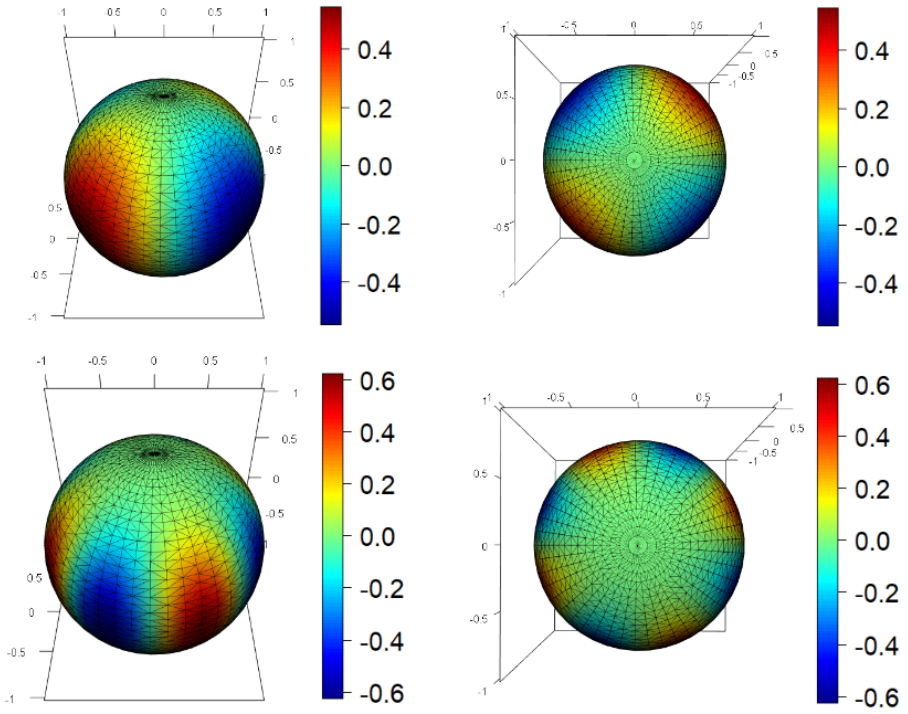

Figure 9: Top: two views of first principal component function v_1; Bottom: two views of second principal component function v_2;

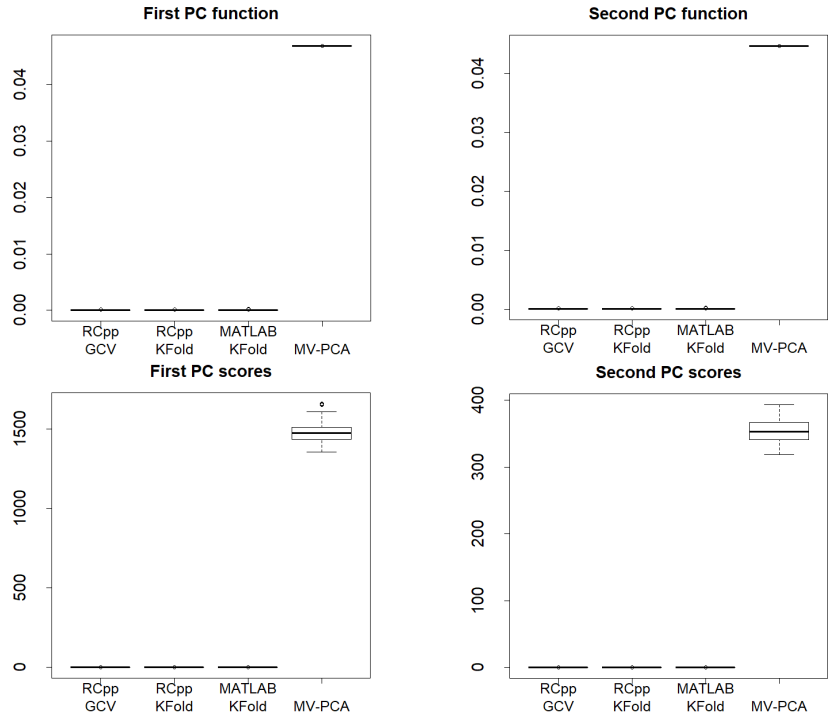

Figure 10: Boxplots summarizing the accuracy performance of MV-PCA and Smooth Functional PCA

PCA algorithms. The MSE has been computed on 100 different datasets, all generated using the formula (35). The results are summarized in Figure 10:

As we can see from the boxplot, the mean square error of the MV-PCA algorithm in both the PC functions and scores is very high compared to the Smooth Functional Principal Component Analysis algorithm, in both of its implementations. The difference is of the order of 10^5. This shows that the proposed algorithm does a very good job in recovering a signal that is distributed on a planar surface or on a manifold.

In order to better appreciate the differences between the `R/C++` implementation and the `MATLAB` one, I removed from the boxplots the results given by the MV-PCA. The results are in Figure 11

The `R/C++` implementation of the Smooth Functional Principal Component Analysis shows a very good accuracy with both the cross-validation methods(the best one seems to be the K-fold cross-validation), and this new implementation shows an overall better accuracy than the `MATLAB` implementation.

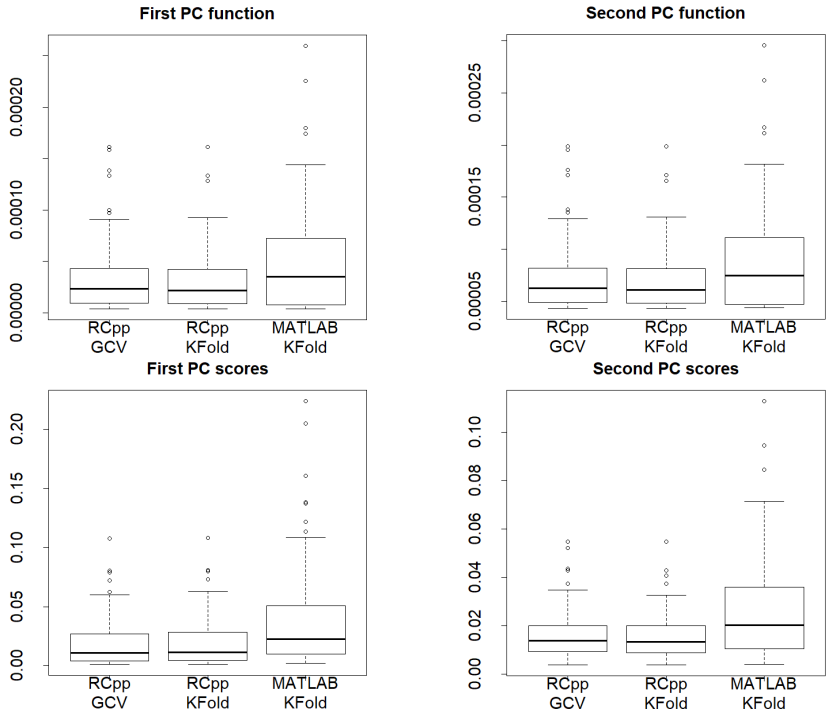

Figure 11: Boxplots summarizing the accuracy performance of the Smooth Functional PCA algorithm in both the `R/C++` and `MATLAB` implementations

4.3 Simulation on the sphere - Computational cost

In this section I will illustrate the speedup performance of the `R/C++` implementation of the Smooth Functional Principal Component Analysis against the `MATLAB` implementation. For this test, I used many triangulated surfaces $\mathcal{M}_{\mathcal{T}}$ that are representations of a sphere of radius $r = 1$ centered in the origin, with only the number of nodes and the number of triangular elements changing. The functional data signals that has been used in these tests have been generated again using the formula (35).For every mesh used, the smooth functions were sampled each time in the locations corresponding to the nodes of the mesh.

The first comparison that has been done between the `R/C++` implementation and the `MATLAB` one is a comparison of the timings of the algorithm with the smoothing λ parameter fixed and without performing any cross-validation. Different number of nodes of the spherical mesh have been used, ranging from 92 to 4832 and also different number of functional data signals included in the datamatrix have been considered, ranging from 50 to 5000 observations. The results of the test are shown in Figure 12.

As we can see, the new `R/C++` implementation shows a significant speedup in the timings of the algorithm, especially when the number of functional data signals in the datamatrix gets bigger. The `R/C++` is in general 3 times faster than the `MATLAB` implementation and is approximately 4 times faster in the case with 4832 nodes and 5000 observations. This is due to the fact that this new implementation uses a compiled source code, that makes everything faster compared to a `MATLAB` implementation.

The next test has been done comparing the times measured for running the SF-PCA algorithm in the `R/C++` implementation of the K-fold cross-validation, the `R/C++` implementation of the GCV and the `MATLAB` implementation of the K-fold cross-validation. This time, the number of functional signals in the dataset was fixed at 50, but, as before, different meshes with varying number of nodes have been considered. The results are shown in Figure 13

The time difference between the `MATLAB` code and the `R/C++` one is quite significant and becomes substantially bigger as the number of nodes of the mesh increases.

Given the fact that for this test I used a small data matrix, with 50 test smooth functions, we see that the K-fold cross validation method is a little bit faster than the GCV method. This is because the K-fold cross-validation is performed by splitting the dataset by rows, while the GCV correspond to a leave-one-column-out validation. Since the dataset taken into account for this test has substantially more columns than rows, better performances of the K-fold algorithm was to be expected.

In the next test I wanted to compare the computational performance of the different cross-validation algorithms by varying the size of the data matrix (the number of smooth functions generated). I used again different meshes representing the sphere of radius $r = 1$ and for each mesh, I timed the execution of the algorithm considering different numbers of samples of the test function. The test was done for both the `R/C++` different versions and the `MATLAB` implementation.

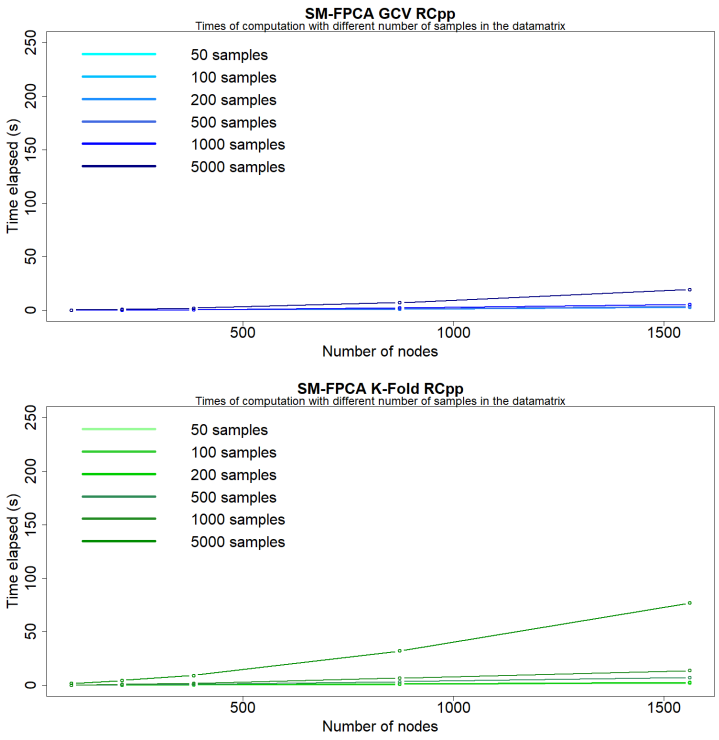

Figure 12: Time comparison of the Smooth Functional Principal Component Analysis algorithm in the `R/C++` and `MATLAB` implementation with a fixed lambda and the number of nodes of the mesh and the number of samples in the dataset varying. The line plotted is the average of 10 executions

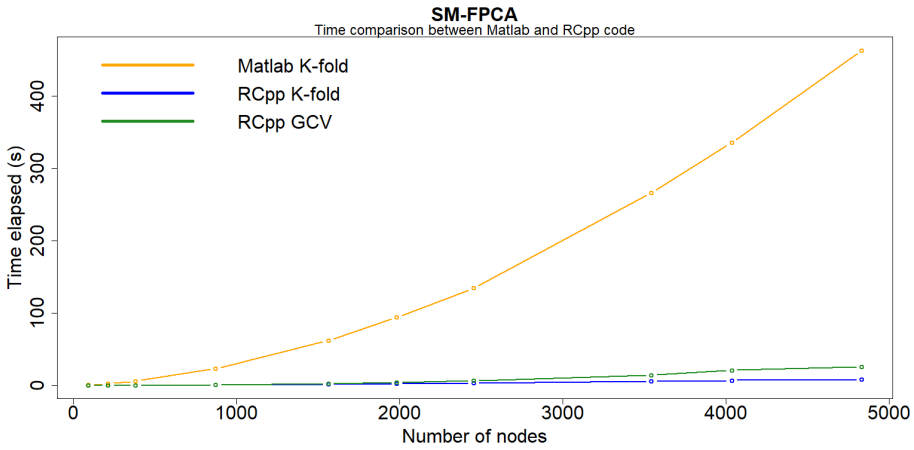

Figure 13: Time comparison of the Smooth Functional Principal Component Analysis algorithm in the `R/C++` and `MATLAB` implementation with the number of nodes of the mesh varying. The line plotted is the average of 10 executions

In Figure 14 we can see that the time of the execution of the `MATLAB` code explodes when we both increase the number of sample functions in the dataset and the number of nodes of the mesh; the computational cost of the `MATLAB` code is not comparable to the computational cost of the `R/C++` implementation. Regarding this implementation, this time we see that with a bigger number of samples in the dataset, the cross validation method that is computationally faster is the GCV method, which do not show a big increase in the computational cost, even with a dataset with 5000 samples. The K-fold cross validation method, otherwise, becomes slower and slower as the number of samples in the dataset increases.

5 Application

As a real case scenario, I applied the proposed Smooth Functional Principal Component Analysis to Neuroimaging data. The dataset considered arises from the Human Connectome Project Consortium and among all the various preprocessing pipelines applied to the HCP original data, the one of particular interest was the *fMRISurface*. This pipeline provides a transformation of the 3D structural MRI and 4D signal from the functional MRI scan in order to enable the application of statistical analysis on the brain surfaces. For each person, the cortical surface is extracted and aligned to a template cortical surface, represented as two triangulated surfaces with 32k nodes each, one for each hemisphere of the brain. To each vertex of the mesh, for each patient, is associated a BOLD time-series derived from the BOLD signal of the underlying gray-matter ribbon. The fMRI used for this analysis has been taken in absence of any task, and thus

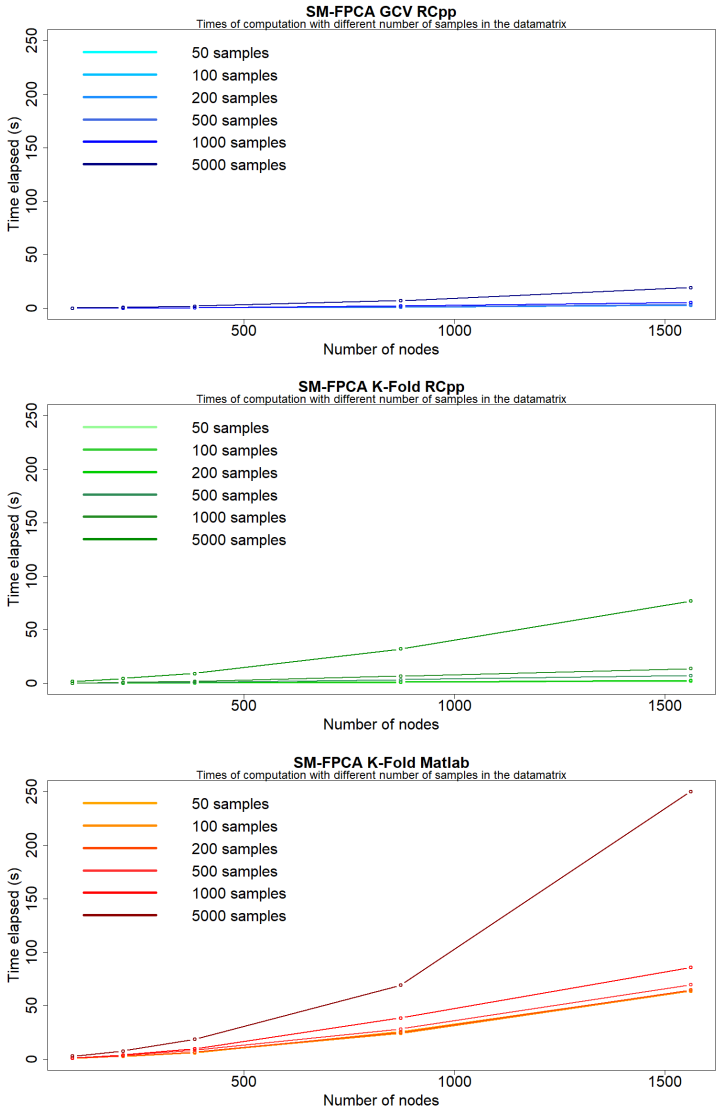

Figure 14: Time comparison of the Smooth Functional Principal Component Analysis algorithm in the `R/C++` and `MATLAB` implementation with the number of nodes of the mesh and the number of samples in the dataset varying. The line plotted is the average of 10 executions

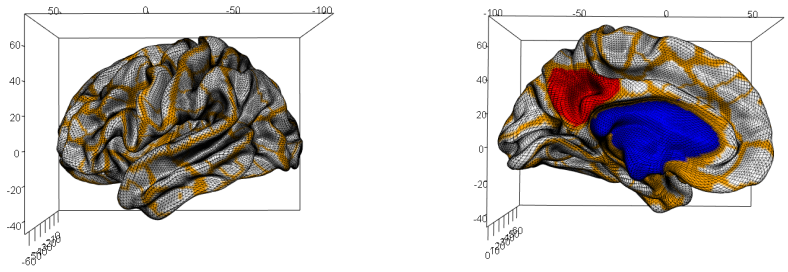

Figure 15: Two views of the triangulated surface with 32k nodes representing the left hemisphere of the brain. The red color indicates the ROI; the blue color indicates the area representing the join between the two hemispheres and therefore does not lie on the manifold; the orange color represent the separations between the parcellated regions

it is called resting state fMRI. In order to study the resting state fMRI, we need to exploit the time dimension of the data and extract a connectivity map among the different parts of the cortical surface. A standard choice for doing this is the computation of the temporal correlation. We identify some region in the brain as the Region of Interest(ROI) on the cortical surface. Within each subject, a cross-sectional average of all time-series in the ROI is used to find a representative mean time-series. To each vertex of the surface, we then associate the pairwise correlation of the time-series located in that vertex with the representative time-series of the ROI. Finally, each value is tranformed using a Fisher's r-to-z transformation, yielding a resting state functional connectivity map(RSFC) for each subject. For the choice of the ROI, we consider the cortical parcellation derived in [6], where a group-average boundary map of the cortical surface is derived from resting state fMRI.

For the application, the triangulated surface taken in consideration was the one relative to the left hemisphere of the brain. The triangulated mesh, with the chosen ROI and the separation between the parcellated regions highlighted, are showed in Figure 15

An example of a mean RSFC map for a patient is showed in Figure 16

We now want to understand the main modes of variation of these RSFC maps among the different subjects. In order to do this, we apply the Smooth Functional PCA algorithm. The dataset used as a starting point for the algorithm contains the RSFC maps of the 71 patients taken into account. The resulting PC functions estimated using the proposed algorithm are showed in Figure 17

The first three PC functions here represented were found using the Smooth Functional Principal Component Analysis algorithm, with the selection of the best λ done with the GCV cross-validation method. As we can see, even if the brain has a complex and very convoluted geometry, the single observations are

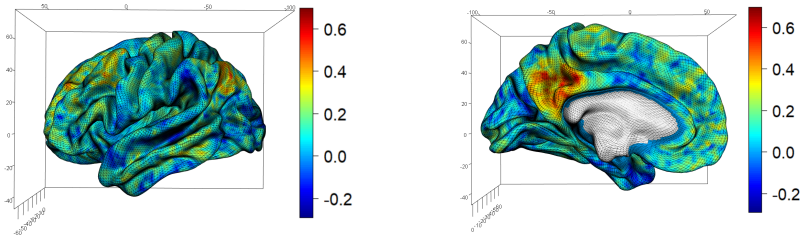

Figure 16: Two views of the RSFC map of one subject. As expected, the map has high correlation values inside the ROI.

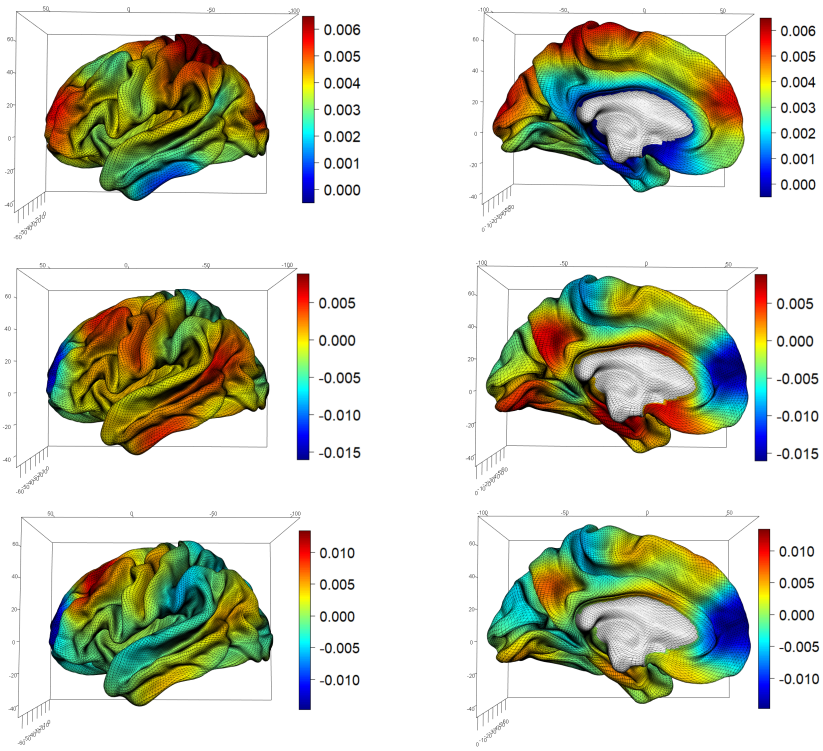

Figure 17: PC function computed using SF-PCA.Top:two views of the first PC function. Middle:two views of the second PC function. Bottom:two views of the third PC function.

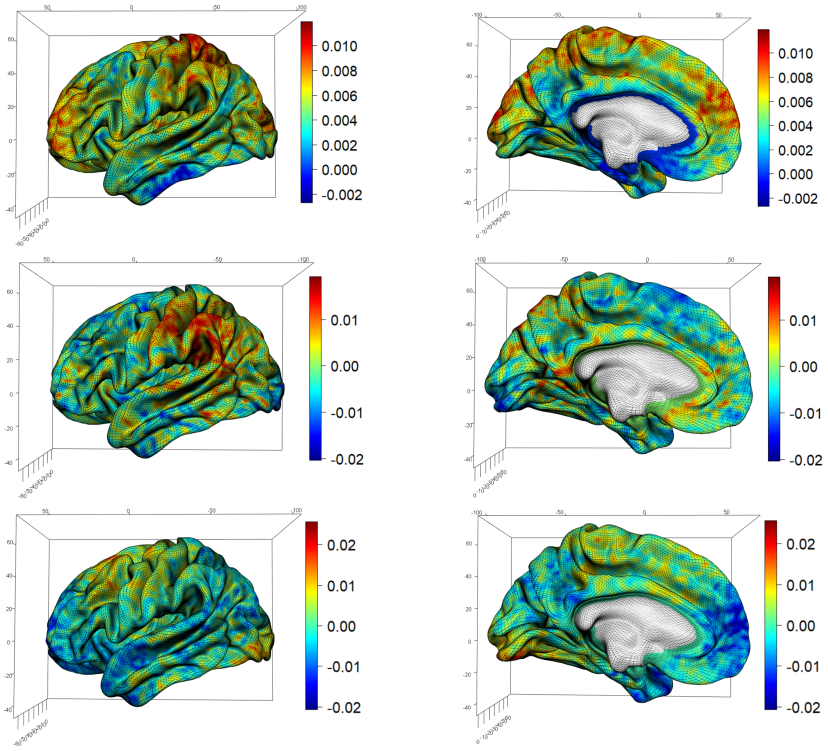

Figure 18: PC function computed using MV-PCA.Top:two views of the first PC function. Middle:two views of the second PC function. Bottom:two views of the third PC function.

noisy and there is the problem of data misalignement due to the registration of the functional data of the subjects to the common template, the PC functions found by the algorithm are very smooth and are able to reduce the variability effects that are present due to misalignement. By looking at the principal components, we can understand in which regions of the brain there is high variability in the RSFC map between the patients. High correlations value are in the regions near the ROI and in the inferior parietal and this is due to individual inter-subject differences.

The estimated first three PC functions using the MV-PCA technique are shown in Figure 18. As we can see, the functions are very irregular and noisy and show an excessive variability, since the sample size is not sufficiently large to deal with the extremely high dimensionality of the data, and the spatial information is completely ignored by this model. The spatial mismatches introduced by the alignement of the functions to a common template represent one of the biggest sources of observable differences between subjects and with this algorithm we are not able to reduce the variability introduced by it.

The SF-PCA algorithm represents a big improvement with respect to the classical PCA technique because is able to capture the localized features of the estimated PC functions while also returning a smooth function that reduces the variability due to misalignement.

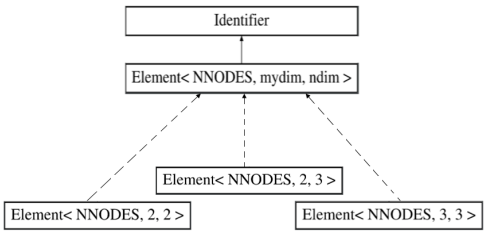

Figure 19: Inheritance graph for the class Element (Doxygen)

6 Ongoing development

The model presented in section 2.2 can be easily extended also to data that is distributed over volumetric domains. Currently there are no techniques able to perform PCA over volumetric domains and this will represent an innovative technique that can be applied in many fields. The development of an extension of the "fdaPDE" library in order to make it able to perform penalized regression models and Smooth Functional Principal Component Analysis over volumetric domains will represent the topic of a Master Thesis and is still an ongoing process, but here in this section I will present some code structures and implementations that I have already done.

The fundamental part of this extension is the implementation of volumetric Finite Elements that are needed for computing the Mass and the Stiffness matrices, used for the assembling of the linear system of both the regression model and the PCA model. For the volumetric Finite Elements, also classes that are able to handle tetrahedral mesh and, more generally, tetrahedral objects are needed.

6.1 The class for Tetrahedra

In order to implement the class able to handle tetrahedron objects and to be consistent with the rest of the code, the `Triangle` class has been renamed and it has become the `Element` class. In this way, we can create the class able to handle tetrahedra by specializing the template parameters `mydim` and `ndim` with the values `mydim=3` and `ndim=3` and keeping the old template specializations untouched. The inheritance graph of the Element class is represented in figure 19.

The structure of the template specialization of class `Element` is here showed.

Listing 11: Template specialization of class `Element` for handling tetrahedra

```
template <UInt NNODES>
class Element<NNODES,3,3> : public Identifier {
public:
    static const UInt numVertices=4;
    static const UInt numSides=3;
    static const UInt myDim=3;
    Real getDetJ() const {return detJ_;}
    const Eigen::Matrix<Real,3,3>& getM_J()
    const {return M_J_;}
    const Eigen::Matrix<Real,3,3>& getM_invJ()
    const {return M_invJ_;}
    Real getVolume() const{return Volume_;};
    Eigen::Matrix<Real,4,1> getBaryCoordinates(const Point& point) const;
    bool isPointInside(const Point& point) const;
    void computeProperties();
    ...
};
```

The methods of this class computes all the geometric properties, like the volume and the barycentric coordinates, that are needed for the Finite Element computation.

6.2 Finite Element and Assembler classes

The next step is the implementation of the volumetric Finite Element class. This implementation has been done again by adding a template specialization of the original `FiniteElement` class.

Listing 12: Header of `FiniteElement` class

```
template <class Integrator ,UInt ORDER, UInt mydim, UInt ndim>
class FiniteElement{
};
```

The `Finite Element` specialization is here showed.

Listing 13: `Finite Element` template specialization

```
template <class Integrator ,UInt ORDER>
class FiniteElement<Integrator, ORDER, 3,3>{
private:
    Element<6*ORDER-2,3,3> reference_;
    Element<6*ORDER-2,3,3> t_;
    Eigen::Matrix<Real,6*ORDER-2, Integrator::NNODES>
     phiMapMaster_;
    Eigen::Matrix<Real,6*ORDER-2, Integrator::NNODES*3>
     phiDerMapMaster_;
```

```
    Eigen::Matrix<Real,6*ORDER-2, Integrator::NNODES*3>
     invTrJPhiDerMapMaster_;
    Eigen::Matrix<Real,3,3> metric_;
    void setPhiMaster();
    void setPhiDerMaster();
    void setInvTrJPhiDerMaster();
public:
    ...
};
```

The first template parameter of the `Element` objects that are included in the `FiniteElement` class is `6*ORDER-2` in order to allow the number of nodes to be 4 in case of first order Finite Element and 10 in case of second order Finite Element, but only the version with `ORDER=1` has been implemented so far.

Every `FiniteElement` object contains two `Element` objects. The two `Element` object are the reference tetrahedron and one of the tetrahedra in the mesh. This is because almost any operation needed to solve our problem is done mapping the generic tetrahedron we are working on to the reference one.The physical tetrahedron is then uploaded by the method `FiniteElement<Integrator, ORDER,3,3>::updateElement` during the iterations over the elements of the mesh.

The other methods of this class allow us to return all the quantities required for the iteration, in particular the evaluations of the basis functions and their derivatives on quadrature nodes. The methods responsible for the evaluation are `phiDerMapMaster_` and `invTrJPhiDerMapMaster_`.

For the assembling of the matrices, the technique used is the one described in Di Pietro e Veneziani [8], where Expression Templates are used to implement Galerkin methods.They are implemented in the file "matrix_assembler.h", where the class `Assembler` and the classes responsible for computing the differential operators are defined. Inside the `Assembler` class, a template specialization has been done for the methods `operKernel` and `forcingTerm`. The class assembler is here showed

Listing 14: `Assembler` template specialization

```
class Assembler{
public:
    template<UInt ORDER, typename Integrator,typename A>
    static void operKernel(EOExpr<A> oper,
    const MeshHandler<ORDER,3,3>& mesh,
    FiniteElement<Integrator, ORDER,3,3>& fe, SpMat& OpMat);
};
```

The `Assembler` class discretizes the generic differential operator defined by class `A`. In particular the method `operKernel` translates the finite element resolution into an algebraic system, defined through the template operation `A`. This structure allows a simple definition of the different types of matrices we use, and

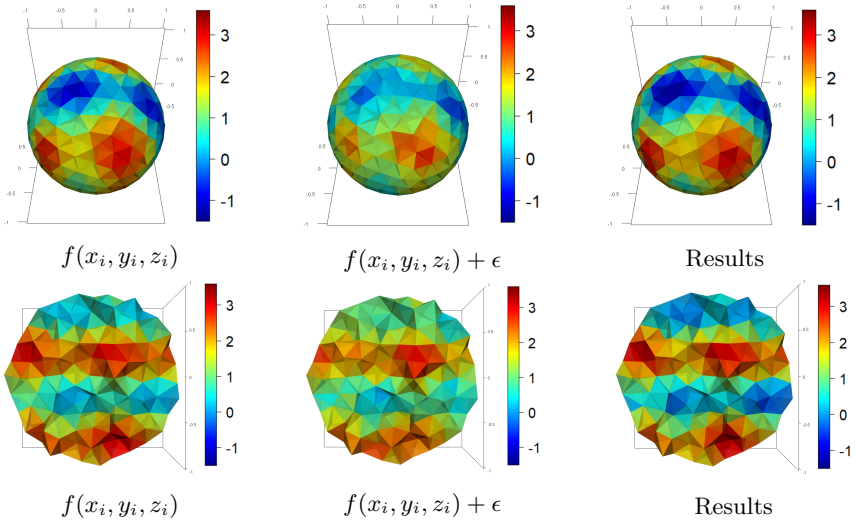

Figure 20: Example of 3D smoothing. Top: outside surface. Bottom: view of a slice inside of the volume

eventually others if required in the future. For our purposes, class `A` can be either of class `Mass` or `Stiff`.

6.3 Smoothing example

With the implementation of these three classes, it is possible to compute the Mass and the Stiffness matrix that are needed for the resolution of the linear system. It is thus possible to test this code on a toy example: I considered a tetrahedral mesh of a sphere, made of 587 nodes and 2775 tetrahedral elements.

A smooth function that assumes values on the nodes of the mesh has been generated according to the formula

$$f(x_i, y_i, z_i) = a_1 \sin(2\pi x_i) + a_2 \sin(2\pi y_i) + a_3 \sin(2\pi z_i) + 1$$

where the terms $a_i, \quad i = 1, 2, 3$ are generated as $a_i \sim N(1, 1)$. In correspondence of the nodes of the mesh a Gaussian white noise ϵ with standard deviation $\sigma = 0.5$ has been added to the true smooth function.

I applied the penalized spatial regression model with the aim to recover the original true function. The true function, the perturbed data and the estimate provided by the spatial regression model are showed in Figure 20.

As we can see from the plot, the spatial regression model is able to recover the true function with a great accuracy on both the inside and the outside of the volumetric object.

Further extensions and many application of this work will be developed in a Master thesis.

7 Installation

The source code of this R package can be found and downloaded from https://github.com/NegriLuca/fdaPDE-manifold. There are two ways of installing it:

- in R, using the popular R package devtools, you can install the fdaPDE package and all of its dependencies using the command

  ```
  install_github("NegriLuca/fdaPDE-manifold")
  ```

  ;using devtools you can also install the package from the terminal. From the root folder, the folder in which you have downloaded and unzipped the source code, you need to run the following command:

  ```
  R -e "library(devtools);  install()" --silent
  ```

  To build the documentation of the package in Roxygen, the command is:

  ```
  R -e "library(devtools);  document()" --silent
  ```

- the second way of installing the package does not require the installation of any other packages, but it will throw an error if the packages required for the functioning of fdaPDE library are not installed. From the terminal, you need to run:

  ```
  R CMD BUILD <path to folder fdaPDE>
  R CMD INSTALL -l <path name of the R library tree>
  <path name of the package to be installed>
  ```

To test the succesful installation of the package, it is possible to run one of the examples in the subdirectory *tests* either from the terminal using

```
Rscript <chosen-test.R>
```

or directly from R.

To build the Doxygen documentation of the C++ code, you need to go to the *src* folder and type from the terminal

```
doxygen -g <config-file>
```

This will create a configuration file that can be edited to customize the output. Once edited, you need to run Doxygen with the command

```
doxygen <config-file>
```

This command will generate a subfolder named *latex*. In this folder you need to type make in order to create the manuals.

7.1 Installing R packages

It may be possible that in Ubuntu or other linux distributions to have troubles and problems in installing the R packages needed for the fdaPDE library, in particular rgl and devtools. If an error of this type

```
checking for X... no
configure: error: X11 not found but required, configure aborted.
```

or this type

```
If instead the error is:
checking GL/glu.h usability... no
checking GL/glu.h presence... no
checking for GL/glu.h... no
configure: error: missing required header GL/glu.h
```

arises during the installation of rgl, it can be fixed by installing

```
xorg-dev libx11-dev mesa-common-dev libglu1-mesa-dev
```

For the package devtools, if it happens an error like

```
ERROR: dependency 'curl 'is not available for package
'httr'
ERROR: dependencies 'httr', 'memoise' are not available
for package 'devtool'...
Warning messages:
1: In install.packages("devtools"):
installation of package 'httr' had non-zero exit status
2: In install.packages("devtools"):
installation of package 'devtools' had non-zero exit status
```

it may be solved by installing the following libraries: `sudo apt-get install libcurl4-openssl-dev libssl-dev` in Ubuntu and `sudo yum -y install libcurl libcurl-devel` in CentOS.

Other packages needed for the library are the R packages plot3D and plot3Drgl.

References

[1] Eardi Lila, Laura M. Sangalli, Jim Ramsay and Luca Formaggia (2017). fdaPDE: Functional Data Analysis and Partial Differential Equations; Statistical Analysis of Functional and Spatial Data, Based on Regression with Partial Differential Regularizations. R package version 0.1-5.

[2] Sangalli L.M., Ramsay J.O., and Ramsay T.O. *Spatial spline regression models*. Journal of the Royal Statistical Society Ser. B, Statistical Methodology, 75, 4, 681-703, 2013.

[3] Azzimonti L., Sangalli L. M., Secchi P., Domanin M., Nobile F. *Blood flow velocity field estimation via spatial regression with PDE penalization* TechRep. MOX 19/2013, Dipartimento di Matematica, Politecnico di Milano, 2013.

[4] Ettinger B., Perotto S., Sangalli L.M. *Spatial regression models over two-dimensional manifolds*. Biometrika, 103 (1), 71-88, 2016.

[5] Lila E., Aston J.A.D., Sangalli L.M. *Smooth Principal Component Analysis over two-dimensional manifolds with an application to Neuroimaging*. Annals of Applied Statistics, 10 (4), 1854-1879, 2016.

[6] Gordon E.M., Laumann T. O., Adeyemo B., Huckins J. F., Kelley W. M., and Petersen S.E. *Generation and evaluation of a cortical area parcellation from resting-state correlations*. Cerebral Cortex, 2014.

[7] *Writing R Extensions*. https://cran.r-project.org/doc/manuals/R-exts.pdf.

[8] Di Pietro D. A. and Veneziani A. *Expression templates implementation of continuous and discontinuous galerkin methods*. Computing and Visualization in Science, 12(8):421-436, 2009.

[9] Quarteroni A. *Numerical Models for Differential Problems.* 2nd edition,Springer Series MS&A, Vol 8, 2014.

[10] Formaggia L., Saleri F. and Veneziani A. *Solving numerical PDEs: problems, applications, exercises*. Springer Verlag, 2012.