POLITECNICO DI MILANO
Master Degree in Mathematical Engineering
Industrial and Information Engineering
Department of Mathematics

fdaPDE: Smoothing Parameter Optimization through Generalized Cross Validation

054073 - Advanced Programming for Scientific Computing Project Report

Professor: Luca Formaggia
Professor: Carlo De Falco
Tutor: Pasquale Africa
Supervisor: Laura Sangalli
Supervisor: Eleonora Arnone

Giorgio Meretti
Andrea Poiatti

September 8th, 2020
Academic Year 2019-2020

Contents

Introduction

Spatial Regression with PDE Regularization is an innovative class of methods for the analysis of spatial and functional data over complex domains. The accessibility of this approach to the general public is possible by the recent CRAN release of `fdaPDE` 1.0 package for the `R` software.

This project is meant to introduce optimization methods in the code structure, in order to facilitate the automatic selection of the best possible PDE-driven Penalization Parameter, also known as smoothing parameter, according to Generalized Cross Validation (GCV), a statistical loss function. In SR-PDE the smoothing parameter plays a key role, balancing the contribution of the PDE-based model to the data-driven estimate. For this reason, a sensible choice of this term is imperative to produce meaningful results; conversely wrong selections might lead to completely senseless models. The state of the art before our work was based on grid evaluation and required the user to provide a vector of possible smoothing parameters to select - if asked to do so - the best candidate according to GCV. This structure had two main flaws: a novice user might not know how to select a valid grid of candidates unknowingly ending up with wrong estimates, an expert would still need to spend a lot of time in the identification of the true values, taking multiple repetitions of the grid evaluation to ensure correctness. Moreover, the default policy - before our contribution - was based on stochastic GCV computation, a technique we prove being source of possible errors, if not treated properly. Our project evolves consistently from this baseline, taking advantage of automatic optimization methods, addressing the aforementioned criticalities in a broad-based user-friendly approach, to provide an efficient and robust smoothing parameter estimate.

The work in this report is an evolution of the structure available at `https://github.com/fdaPDE/fdaPDE`, the current state of the art of `fdaPDE`. However, having our effort spanned approximately one year, we had the opportunity to migrate across various past versions of the library: from Negri [10], responsible for the introduction of Functional Principal Component Analysis and Regression reorganization, passing through Colli-Colombo [4], whose project has made possible the aforementioned CRAN release, up to Colombo-Perin [5], creditable for having introduced the latest release containing Generalized Additive Models.

The necessity of adapting our work several times and for multiple interfaces has proved a fascinating challenge and has helped us in rationalizing our coding style prioritising portability, efficiency and ease of extension. Nonetheless, the inability to even compile the code on multiple average-RAM machines - due to a series of contributing causes that we will discuss in detail in this report - lead us towards the need of a rational code reorganization, that we performed together with division in sub-directories and de-templatization.

We would like to thank Professor Laura Sangalli and Tutor Eleonora Arnone for the patience and kindness to support us during this journey. A honorable mention also to our colleague Gabriele Gabrielli, who helped us in testing the library compilation on multiple architectures.

Our code is available at `https://github.com/GMeretti/PACS_merettipoiatti`, for correct visualization of the indentation we suggest to use tab size 8, GitHub default. The core part of our work is contained in `src/Lambda_Optimization`.

Brief overview of the content in the project:

- ➔ Chapter 1 is a theoretical introduction to Regression problems with PDE regularization.
- ➔ Chapter 2 explains in detail the new structure we have introduced in directory `src`, de-templatization, code reorganization, cleaning and one correction of a relevant mistake.
- ➔ Chapter 3 contains an overview of Generalized Cross Validation as function of the penalization parameter λ. In particular we introduce our study of the GCV derivatives with an eye on efficient evaluation and storage of the byproducts of the computation. We finally analyze the optimization methods we have deemed useful to be employed for the identification of its minimum.
- ➔ Chapter 4 is a rational description of the code we have written and is meant to explore the optimization process, following its structure, from the `R` user to `C++` and then again back to `R`. In each Section we highlight a method, stressing its role in the pattern and its main peculiarities from a technical standpoint.
- ➔ Chapter 5 collects several simulation studies that corroborate our belief concerning the potential of the methods we have developed.
- ➔ Chapter 6 provides guidance on installation among different architectures and also shares some hints about how to perform tests.
- ➔ Chapter 7 discusses how to comment the code and concludes tracking some possible extensions of our work that could be implemented in the future.

Highlights

The main novelties introduced in our new version of the library can be summarized into two nodal points:

Code reorganization

We adopt a more rational partitioning of the source file `fdaPDE.cpp`, which contained too many functions and led to an excessively high RAM consumption during the library installation: before our version, some PC's with RAM 8GB could not even compile the library. We also separate the files of the `src` directory in sub-directories, in order to give a more clear order to the entire structure of the library, and reduce the over-templatization of some crucial classes.

GCV automatic optimization methods

We introduce the analytical computation of the first and second derivatives of the Generalized Cross Validation (GCV) and we exploit them to implement an exact Newton's method and an approximated method by means of finite differences. We then show by means of some tests, that these methods are more effective in terms of either computational time or precision and stability, with respect to the simple grid evaluation of the GCV. Indeed, in general the user knows nothing about the real minimizer and thus an automatic way of finding this optimal value is the most robust and cheapest choice.

For what concerns the code, by means of variadic templates, we build a template class, called `Carrier`, which is able to treat the different possibilities of smoothing allowed. Then, by the construction of other SFINAE-based template functions, we give the possibility of computing the GCV in all its possible cases, either with exact computation or stochastic. Using another variadic template class, the `Function_Wrapper`, generalizing the concept of *function*, we give the possibility of evaluating a function and its first and second derivatives. Relying on this wrapper, we introduce the aforementioned optimization methods to minimize the GCV.

We improve the `R` interface, by adding much more checks for the inputs coming from the user and all the new features linked to the optimization methods just introduced, and we also reorganize the output returned to `R`, in order to comply with the new functionalities.

Chapter 1

Theoretical Background

In this Chapter we present a class of models that can handle data displaying complex spatio-temporal dependencies, in particular Spatial Regression with Partial Differential Equation regularization and Smooth Functional Principal Component Analysis with Partial Differential Equation regularization. These models are based on the idea of regression with a regularization term that involves a Partial Differential Equation (PDE). The synergistic interplay of approaches from statistics, applied mathematics and engineering endows the methods with important advantages with respect to the available techniques, and makes the methods able to accurately deal with data structures for which the classical methods are unfit.

1.1 Regression model

Let $\mathbf{p}_1, \ldots, \mathbf{p}_n$ be n data locations[1], distributed on a bounded spatial domain $\Omega \subset \mathbb{R}^2$, sufficiently regular (e.g., $\partial\Omega \in C^2$). Let $z_i \in \mathbb{R}$ be the value of a variable of interest observed at $\mathbf{p}_i$, called observation. Furthermore, in case covariates were available, let $w_i \in \mathbb{R}^k$ be the k-dimensional vector of k covariates associated with z_i at the corresponding location. We consider, as in [14], the following semiparametric generalized additive model:

$$z_i = \mathbf{w}_i^T \boldsymbol{\beta} + f(\mathbf{p}_i) + \varepsilon_i, \quad i = 1, \cdots, n \tag{1.1}$$

where $\boldsymbol{\beta} \in \mathbb{R}^k$ is an unknown vector of regression coefficients corresponding to the effects of covariates on the mean value of the variable of interest, $f : \Omega \to \mathbb{R}$ is an unknown deterministic field capturing the spatial structure of the phenomenon under analysis, and $\{\varepsilon_i\}_{i=1}^n$ are random errors with null mean and finite variance. In [15] the authors propose to estimate the unknown $\boldsymbol{\beta}$ and f by minimizing the regularized least-square functional

$$\sum_{i=0}^{n} (z_i - \mathbf{w}_i^T \boldsymbol{\beta} - f(\mathbf{p}_i))^2 + \lambda \int_\Omega (\Delta f)^2 d\mathbf{p} \tag{1.2}$$

with λ seen as a positive smoothing parameter and Δ denoting the two-dimensional Laplacian operator. When covariates are not required, the model reduces to

$$z_i = f(\mathbf{p}_i) + \varepsilon_i, \quad i = 1, \cdots, n \tag{1.3}$$

and the estimation proceeds as before, neglecting the terms related to the covariates $\mathbf{w}_i$.

If we assume that some partial problem-specific information is available, coming, e.g., from the physics, mechanics, chemistry or morphology of the problem at hand, and that this information can be formalized in terms of a PDE $\mathcal{L}f = u$, where $\mathcal{L}$ is a linear differential operator and u is a forcing term - i.e., on the base of the problem-specific information, we can assume that the

[1] In the following we adopt the boldface to indicate the vectors, whereas we use upper case letters for matrices and lower case letters for scalar values.

misfit $\mathcal{L}f - u$ is small, though we do not require it to be null - then, it makes sense to estimate the unknown $\boldsymbol{\beta}$ and f by minimizing the functional

$$\sum_{i=0}^{n}(z_i - \mathbf{w}_i^T\boldsymbol{\beta} - f(\mathbf{p}_i))^2 + \lambda\int_\Omega(\mathcal{L}f - u)^2 d\mathbf{p} \tag{1.4}$$

We will always assume $\mathcal{L}$ being an elliptic differential operator: the most general form of advection-diffusion-reaction considered up to now in the library is the following

$$\mathcal{L}f = -\text{div}(K\nabla f) + \mathbf{b}\cdot\nabla f + cf \tag{1.5}$$

with $K \in \mathbb{R}^{2\times 2}$ an anisotropy matrix, $\mathbf{b} \in \mathbb{R}^2$ a transport vector and $c \in \mathbb{R}$ a reaction factor. All of these coefficients could be space-varying (e.g., $K = K(\mathbf{p})$). For what concerns the boundary conditions, f can satisfy particular problem-specific requirements: we consider Neumann boundary conditions, regarding the conormal derivative, and Dirichlet boundary conditions. If we set $\Gamma_D,\ \Gamma_N \subset \partial\Omega$, $\mathring{\Gamma}_D \cap \mathring{\Gamma}_N = \varnothing$ and $\Gamma_D \cup \Gamma_N = \partial\Omega$ we have:

$$\begin{cases} f = \gamma_D \ \text{ on } \Gamma_D \\ K\nabla f\cdot\nu = \gamma_N \quad \text{ on } \Gamma_N, \end{cases}$$

with ν the outward unit normal vector. For simplicity of exposition, in the following, we will only consider homogeneous Dirichlet or Neumann conditions; see [2] for details.

1.2 Finite Elements approximation

Finite elements are used to approximate the solutions of PDEs, thus are very useful in this context. In general triangular meshes are employed: let $\mathcal{T}$ a triangulation of domain Ω and define Ω_T as the union of the triangles. We can define the finite element space

$$V_\mathcal{T}(\Omega) = \{\mathbf{v}_h \in C^0(\bar{\Omega}) : \mathbf{v}_h|_T \in \mathbb{P}_r, \forall T \in \mathcal{T}\}.$$

with $\mathbb{P}_r$ representing the polynomials of order $r \in \mathbb{N}$.

Defining $\{\boldsymbol{\xi}_i\}_{i=0}^{N_\mathcal{T}}$ as the nodes of the mesh, we can introduce the basis functions $\{\psi_j\}_{j=0}^{N_\mathcal{T}}$, which are Lagrangian functions, i.e. each ψ_j is 1 on the corresponding $\boldsymbol{\xi}_j$ and 0 on the other nodes ($\psi_j(\boldsymbol{\xi}_i) = \delta_{ij},\ \forall i,\ j = 1,\cdots,N_\mathcal{T}$). Let $\boldsymbol{\psi} = \{\psi_1,\cdots,\psi_{N_\mathcal{T}}\}^T$ be the vector of the basis functions. Any function $v \in V_\mathcal{T}(\Omega)$ can be represented as

$$v(\mathbf{p}) = \mathbf{v}^T\boldsymbol{\psi}(\mathbf{p})$$

where $\mathbf{v} \in \mathbb{R}^{N_\mathcal{T}}$ is the vector of the projections of v onto the basis functions. Due to the Lagrangian property of the basis functions, we have that $v_i = v(\boldsymbol{\xi}_i),\ \forall i = 1,\cdots,N_\mathcal{T}$.

1.3 Solution of the estimation problem

Let Ψ be the $n \times N_\mathcal{T}$ matrix evaluating the basis functions previously defined at the n data locations $\mathbf{p}_1,\ldots,\mathbf{p}_n$:

$$\Psi = \begin{bmatrix} \psi_1(\mathbf{p}_1) & \cdots & \psi_{N_\mathcal{T}}(\mathbf{p}_1) \\ \vdots & \vdots & \vdots \\ \psi_1(\mathbf{p}_n) & \cdots & \psi_{N_\mathcal{T}}(\mathbf{p}_n) \end{bmatrix} \tag{1.6}$$

Moreover let R_0 and R_1 be the following $N_\mathcal{T} \times N_\mathcal{T}$ matrices:

$$R_0 = \int_{\Omega_\mathcal{T}} \boldsymbol{\psi}\boldsymbol{\psi}^T \qquad R_1 = \int_{\Omega_\mathcal{T}} \nabla\boldsymbol{\psi}^T K\nabla\boldsymbol{\psi} + \nabla\boldsymbol{\psi}^T\mathbf{b}\boldsymbol{\psi} + c\boldsymbol{\psi}\boldsymbol{\psi}^T \tag{1.7}$$

We can identify three cases of interest:

- Isotropic and stationary case: K multiple of the identity, the advective and reaction terms are null, $u = 0$.
- Anystostropic and stationary case: K not a multiple of the identity, $\mathbf{b}$ and c possibly nonvanishing, $u = 0$.
- Anysotropic and nonstationary case: K, $\mathbf{b}$ and c may be space varying, and also the forcing term u could be added.

In order to introduce the regression setting, we consider the following matrices related to the estimation problem: we start from W, the covariate matrix:

$$W = \begin{bmatrix} \mathbf{w}_1^T \\ \vdots \\ \mathbf{w}_n^T \end{bmatrix},$$

Note that each column of W represents the evaluation of the respective covariate (also said regressor) in the n locations considered. We then introduce the projection matrix Q which is the projector onto the $\mathbb{R}^n$ orthogonal complement of the k-dimensional subspace spanned by the columns of W:

$$Q = I - W(W^TW)^{-1}W^T$$

and we also define $H = I - Q$, the projection onto the the aforementioned k-dimensional subspace generated by the columns of W.

The following theorem states that the estimation problem, in the Finite Elements setting, reduces to the following linear system to be solved, considering the case when $u = 0$, i.e., without forcing term:

Theorem 1.3.1. There exists a unique pair of estimators $(\hat{\boldsymbol{\beta}} \in \mathbb{R}^k, \hat{f}_{\mathcal{T}} \in V_{\mathcal{T}}(\Omega))$ which solves the discretized estimation problem. Furthermore,

$$\hat{\boldsymbol{\beta}} = (W^TW)^{-1}W^T(\mathbf{z} - \Psi\hat{\mathbf{f}})$$

with $\mathbf{z}$ as the vector of observations, and $\hat{f}_{\mathcal{T}} = \hat{\mathbf{f}}^T\boldsymbol{\psi}$, with $\hat{\mathbf{f}}$ satisfying

$$\begin{bmatrix} -\Psi^TQ\Psi & \lambda R_1^T \\ -\lambda R_1 & -\lambda R_0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{f}} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} -\Psi^TQ\mathbf{z} \\ \mathbf{0} \end{bmatrix}, \tag{1.8}$$

System 1.8 is of order $2N_{\mathcal{T}}$, recalling that usually $n \leq N_{\mathcal{T}}$ (and often $\ll$) we can assume it being very large, in general. On the other hand, it is very sparse, due to the presence of the matrices R_0 and R_1 and the Lagrangian basis functions. From Theorem 1.3.1 we deduce that $\hat{f}_{\mathcal{T}} = \hat{\mathbf{f}}^T\psi$, with $\hat{\mathbf{f}} = (\Psi^TQ\Psi + \lambda R)^{-1}\Psi^TQ\mathbf{z}$, where the positive definite matrix $R := R_1^TR_0^{-1}R_1$ represents the discretization of the penalty term in (1.4). Clearly, when the covariates are not included in the model, the matrix Q is the identity.

This regression model can be extended also to data distributed over Riemannian manifolds and volumes (see [6] and [9]) with the exception that only isotropic stationary regularization can be performed up to now (model (1.2)). For what concerns the possibility of considering a forcing term, we can consider a function $u \in L^2(\Omega)$, which has to be evaluated at the mesh points, to get the vector $\mathbf{u} = [u(\boldsymbol{\xi}_1), \ldots, u(\boldsymbol{\xi}_{N_{\mathcal{T}}})]^T$. Then, the solution of the discretized estimation problem becomes

$$\begin{bmatrix} -\Psi^TQ\Psi & \lambda R_1^T \\ -\lambda R_1 & -\lambda R_0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{f}} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} -\Psi^TQ\mathbf{z} \\ -\mathbf{u} \end{bmatrix}, \tag{1.9}$$

whereas all the other equations remain unchanged. In upcoming Section 1.4 we deal with a variant of the model: the presence of areal data. After this presentation, we will describe the method of estimating the penalization parameter λ by means of the Generalized Cross Validation, which is the main topic of our project.

1.4 Regularized regression model for areal data

If data are no longer observed at pointwise locations $\mathbf{p}_1, \dots, \mathbf{p}_{N_\mathcal{T}}$, but instead on n disjoint subdomains of the primary Ω: $D_1, \dots, D_n$. Let $z_1, \dots, z_n$ be the mean value of $z(\mathbf{p}) \in L^2(\Omega)$ over $D_i,\ i = 1, \dots, n$. The following model is considered:

$$z_i = \mathbf{w}_i^T \boldsymbol{\beta} + \frac{1}{|D_i|} \int_{D_i} f d\mathbf{x} + \varepsilon_i \tag{1.10}$$

where $\varepsilon_1, \dots, \varepsilon_1$ are random errors with variance $Var(\varepsilon_i) \propto \frac{1}{|D_i|}$. Therefore, following the reasoning as in the previous Sections, in [2], the authors propose the estimation of f and $\boldsymbol{\beta}$ by minimizing the functional, which already takes into account the covariates and forcing term u:

$$\bar{J}_{\lambda,\mathcal{L},u}(\boldsymbol{\beta}, f) = \sum_{i=1}^{n} |D_i| \left(\mathbf{z}_i - \mathbf{w}_i^T \boldsymbol{\beta} - \frac{1}{|D_i|} \int_{D_i} f d\mathbf{x} \right)^2 + \lambda \int_\Omega (\mathcal{L} f - u)^2 d\mathbf{p} \tag{1.11}$$

over $\boldsymbol{\beta} \in \mathbb{R}^k$. We recall that the sum is weighted by $|D_i|$, since $Var(\bar{\varepsilon}_i) \propto \frac{1}{|D_i|}$.

After the Finite Element discretization, the authors in [2] show that the following theorem holds for model (1.10):

Theorem 1.4.1. There exists a unique pair of estimators $(\hat{\boldsymbol{\beta}} \in \mathbb{R}^k, \hat{f}_\mathcal{T} \in V_\mathcal{T}(\Omega))$ which solve the discretized estimation problem. Furthermore,

$$\hat{\boldsymbol{\beta}} = (W^T W)^{-1} W^T (\mathbf{z} - \Psi \hat{\mathbf{f}})$$

with $\mathbf{z}$ as the vector of observations, and $\hat{f}_\mathcal{T} = \hat{\mathbf{f}}^T \boldsymbol{\psi}$, with $\hat{\mathbf{f}}$ satisfying

$$\begin{bmatrix} -\Psi^T A Q \Psi & \lambda R_1^T \\ -\lambda R_1 & -\lambda R_0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{f}} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} -\Psi^T A Q \mathbf{z} \\ -\lambda \mathbf{u} \end{bmatrix}, \tag{1.12}$$

with $\Psi = \begin{bmatrix} \frac{1}{|D_1|}\int_{D_1} \psi_1 & \cdots & \frac{1}{|D_1|}\int_{D_1} \psi_{N_\mathcal{T}} \\ \vdots & \ddots & \vdots \\ \frac{1}{|D_n|}\int_{D_n} \psi_1 & \cdots & \frac{1}{|D_n|}\int_{D_n} \psi_{N_\mathcal{T}} \end{bmatrix}$ and $A = \begin{bmatrix} |D_1| & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & |D_n| \end{bmatrix}$ incidence matrix.

The innovations that we brought to the theory will be addressed, together with the definition of Generalized Cross Validation, in Chapter 3. In particular, we will describe how to compute the first and second derivatives of these functions, in order to use them in an optimization method.

Chapter 2

Code Reorganization

This Chapter aims to describe the main steps we followed in order to rationalize the shape of the `C++` code in the `fdaPDE` library. As will become more evident afterwards, the main idea behind this refactoring was the prospect of addressing - and possibly solving - some of most renowned issues concerning the code structure and subsequent compilation. The discussion will focus on the changes that have been performed, with an eye on both their reasons and the way they proved useful from a user and/or a developer standpoint. In particular, we hope that this part of our work will improve the usability for future contributors of the library.

2.1 State of the art drawbacks

As already specified in the introduction given the temporal extension of our project, we have enjoyed the opportunity to experiment first hand both the pros and cons of many versions of the `fdaPDE` libraries. Focusing on the second category, the swift increase of material in the package has lately resulted in an even more appreciable exacerbation of some criticalities that have accompanied the code structure, with growing impairment, since its birth. We can identify three main units, which will be described afterwards. We will dwell on the technical details and attempted corrections concerning these topics in the next Sections.

1. **Overtemplatization**: The `C++` code makes a rather intensive usage of the template construct. This fact - *per se* - is not detrimental, enabling both structural flexibility and speeding up computations at running time. However, template instantiation is notoriously more demanding at compile time, thus templatization should always be tempered by pertinence. Form a purely formal perspective, we might be tempted to state that it is usually inefficient employing an unnecessary template parameter in a class. This because, as the term will assume multiple values, several copies of the same code will be performed without noticeable gain. Unfortunately, this is exactly the case in some classes of the `fdaPDE`, and the consequential loss in time gains moderately relevant.

2. **Compilation units coagulation**: Flowing across the history of this library, one might note that source file `fdaPDE.cpp` has traditionally been the sole collector of all the functions that share direct connection with `R` code. This choice allows to fasten compilation to some extent, but - besides the readability reduction - together with over-templatization is a major cause of an extensive RAM usage per compilation. To give an idea of the severity of the threat represented by this issue: the memory exhaustion is so massive to completely freeze an 8 GB machine during a compilation attempt.

3. **Chaotic source management**: Even though not affecting directly the user, this problem is a remarkable inconvenience for developers. Having the library grown to the size of almost one hundred `C++` files, the lack of a proper partitioning in meaningful subdirectores, makes browsing across sources quite troublesome. Also the lack of a common syntax in the nomenclature of the various units penalises the eye-catching factor and - by reiteration -

the overall comprehension of the code structure. This drawback strikes with more intensity a novice developer that approaches the code for the first time, making quite difficult to find its vital nodes. Clearly, this problem has grown more and more relevant, given the wide variety of features currently supported by the library.

2.2 Rational partitioning of `fdaPDE.cpp`

We start by describing in which way we decided to split the `fdaPDE.cpp`, according to its content. Note that version [5] of the code presented the following code structure:

1. sorting functions to be called via `.call()` by `R` code, we enumerate them in a meaningful order in the following, providing a short description of their purpose:
   I. `regression_Laplace`: manages spatial regression problems with Poisson-Dirichlet equations, forcing term not allowed, areal data allowed;
   II. `regression_PDE`: manages spatial regression problems with constant coefficients elliptic Dirichlet equations, forcing term not allowed, areal data allowed;
   III. `regression_PDE_space_varying`: manages spatial regression problems with space dependent elliptic Dirichlet equations, forcing term allowed, areal data allowed;
   IV. `regression_Laplace_time`: manages spatio-temporal regression problems with Heat-Cauchy-Dirichlet equations, forcing term not allowed, areal data allowed;
   V. `regression_PDE_time`: manages spatio-temporal regression problems with constant coefficients parabolic Cauchy-Dirichlet equations, forcing term not allowed, areal data allowed;
   VI. `regression_PDE_space_varying_time`: manages spatio-temporal problems with space dependent parabolic Cauchy-Dirichlet equations, forcing term allowed, areal data allowed;
   VII. `GAM_Laplace`: manages spatial exponential family generalized regression problems with Poisson-Dirichlet equations, forcing term not allowed, areal data allowed;
   VIII. `GAM_PDE`: manages spatial exponential family generalized regression problems with constant coefficients elliptic Dirichlet equations, forcing term not allowed, areal data allowed;
   IX. `GAM_PDE_space_varying`: manages exponential family generalized spatial regression problems with space dependent elliptic Dirichlet equations, forcing term allowed, areal data allowed;
   X. `Smooth_FPCA`: manages functional principal component analysis, areal data allowed;
   XI. `Density_Initialization`: manages initialization of problems of density estimation;
   XII. `Density_Estimation`: manages evaluation of problems of density estimation;
   XIII. `get_integration_points`: a function required for anysotropic and non-stationary regression (only 2D);
   XIV. `get_FEM_mass_matrix`: a utility returning mass matrix, not used for system solution, may be used for debugging;
   XV. `get_FEM_stiff_matrix`: a utility returning stiffness matrix, not used for system solution, may be used for debugging;
   XVI. `get_FEM_PDE_matrix`: produces problem matrix for elliptic constant coefficients case;
   XVII. `get_FEM_PDE_space_varying_matrix`: produces problem matrix for elliptic space varying case.

2. template functions named skeletons, linking the deeper `C++` levels with sorting functions:
   i. `regression_skeleton`: called by methods I-III, unified treatment of spatial regression problems;
   ii. `regression_skeleton_time`: called by methods IV-VI, unified treatment of spatio-temporal regression problems;
   iii. `GAM_skeleton`: called by methods VII-IX, unified treatment of exponential family generalized regression problems;
   iv. `FPCA_skeleton`: called by method X, unified treatment of functional principal component analysis;
   v. `DE_init_skeleton`: called by method XI unified treatment of density estimation initialization;
   vi. `DE_skeleton`: called by method XII unified treatment of density estimation problems;
   vii. `get_integration_points_skeleton`: called by methods XIII, basic utility for integration points evaluation;
   viii. `get_FEM_Matrix_skeleton`: called by methods XIV-XVII, basic utility for matrix evaluation.

In order to address the problems of excessive memory consumption described in Section 2.1 we have decided to follow the subsequent division of the macro source file, via common content initialization. The following Table 2.1 synthesizes the new arrangement. Please note that regression related functions have been gathered in relation to their `InputHandler`, namely Laplace problems, constant coefficients elliptic PDEs and space varying coefficients elliptic PDEs. This might seem counterituitive at first glance since the grouping involves functions coming from different contexts (spatial, spatio-temporal and GAM regression). However, this division is more sensible for reasons regarding compilation efficiency, since decreases the overall amount of memory and time consumption. Actually, this is a byproduct of the way in which regression data are templatized and stored, depending on the elliptic part of the problem they try to solve.

New src	Old functions	Content
`Rfun_Regression_Laplace.cpp`	I, IV, VII	Laplace problem reg.
`Rfun_Regression_PDE.cpp`	II, V, VIII, XVI	elliptic regression
`Rfun_Regression_PDE_Space_Varying.cpp`	III, VI, IX, XVII	general elliptic reg.
`Rfun_Smooth_FPCA.cpp`	X	functional pca
`Rfun_Density_Estimation.cpp`	XI, XII	density estimation
`Rfun_Auxiliary.cpp`	XIII, XIV	debugging utilities

Table 2.1: New source organization

Parallel to this operation, also skeletons have been provided of an autonomous header (being templates) in which to be stored, according to the criterion expressed in Table 2.2.

New header	Old functions	Content
`Regression_Skeleton.h`	i	spatial regression
`Regression_Seleton_Time.h`	ii	spatio-temporal regression
`GAM_Skeleton.h`	iii	GAM regression
`FPCA_Skeleton.h`	iv	functional pca
`DE_Skeleton.h`	vi	density estimation
`DE_Initialization_Skeleton.h`	v	density initialization
`Auxiliary_Skeleton.h`	vii, viii	debugging utilities

Table 2.2: New skeletons

Note: of course dividing the macro source file `fdaPDE.cpp` in smaller cohesive compilation units has both pros and cons. As can be seen in Figure 2.1, the computational memory requirement

peak is strongly decreased in response to the partitioning, we descent from more than 5.5 GB to 3.5. However, this has a necessary drawback in terms of compile time (still kept under control thanks to de-templatization that we will discuss, in Section 2.4), from 4.5 to almost 10 minutes. As a disclaimer, note that the results are referred to a 16 GB RAM laptop, slightly different results can be obtained changing device. Finally, from a developer viewpoint the partition brings along huge advantages: differently from before, having separate units allows to recompile just the part undergoing changes without having to wait for the whole structure to be rewritten in testing phase.

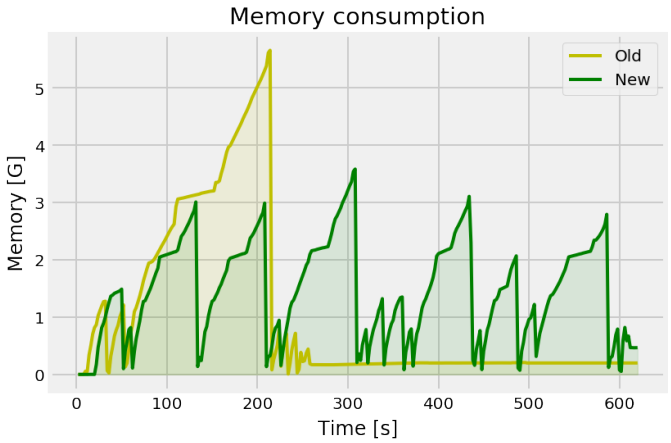


Figure 2.1: Comparison between new and old memory consumption

As a final remark, note that our work has intentionally tried to preserve the two layers structure "sorting function" → "skeleton" as originally conceived in the library. This, of course, for reasons of consistency, but perhaps more for the sake of extensibility. Indeed, whichever the plugins this library will support in the future, new developers are strongly recommended to follow the presented schema, building their own " `R` caller" + "skeleton" and then taking advantage of the common `C++` substructure, thus preserving both symmetry and efficiency.

2.3 Introducing sub-directories and common syntax

In order to overcome the chaotic source management described in the first Section we also decided to give a stylistic imprint to our work, in particular creating a new common syntax for all the source and headers, just to subsequently divide them in sub-directories. We report the chosen fashion as a reference for future workers on the library:

- All letters are lowercase besides the initial of each word and acronyms.
- Different words are separated by an underscore.
- Header files have `.h` extensions, sources `.cpp`.
  As for template implementation, we kept the original `_imp.h` nomenclature.
- `R` called functions start with `Rfun`.
- Header guards are of the form: `__NAME_OF_THE_FILE_H__`.

As an example, consider the following names (belonging to real functions inside the library): `DE_Initialization_Skeleton.h`, `Rfun_Regression_Laplace.cpp`, `Regression_Data_imp.h`. In Appendix A we have reported the full new structure of the `src` directory, sub-divided by content. Each folder we created is partitioned in `Include`, containing header files, and `Source`, for the source code. This division is somehow standard in huge `R` libraries, for some reference see e.g. CRAN package `Matrix` (see also [12]). There are also a few exceptions to the aforementioned rules. For instance, the split is not possible in folder `C_Libraries` containing files

like `triangle.h` [21] that is "freely available, copyrighted by the author and may not be sold or included in commercial products without a license", and cannot be modified and/or renamed. Same for files that are outside any folder and use specific syntax proper of `R` libraries.

2.4 Improving over-templatization

The last topic concerning code reorganization involves templatization review, as we have already pointed out in Section 2.1, `fdaPDE` library presented two main problems with templates that can be summarized in two categories: source code treated as template code and thus put in `.h` files instead of `.cpp`, presence of redundant template parameters in some positions handicapping both readability and efficiency.

As for the first problem, the presence of spurious headers has been a constant factor in the library's history, that we are allowed to trace back since Negri's code [9], but probably was already present in the CRAN version of 2016 or even before. As a matter of facts, this "error" has never been spotted so far as an indirect consequence of the existence of macro-functions like `fdaPDE.cpp` allowing the compilation of the aforementioned classes a single time. With the split of Section 2.2, it has proved necessary to reorganize the code properly, identifying class members definitions in `.h` files and putting them in the proper source. Precisely the corrections, besides some minor changes, have involved mostly three classes (everything is already written with the new syntax):

- `FPCA_Data_imp.h` → `FPCA_Data.cpp`
- `FPCA_Object_imp.h` → `FPCA_Object.cpp`
- `Regression_Data_imp.h` → `Regression_Data.cpp`

Concerning de-templatization, our efforts have mainly regarded file `Mixed_FE_Regression.h` and its FPCA counterpart `Mixed_FE_FPCA.h`. The key factor of this operation has been the proper partition of `apply` function, mimicking its division in the FPCA setting. Until Colombo-Perin [5] the function `apply` in `Mixed_FE_Regression.h` has always been the common template macro function which first set all the matrices relevant for any regression problem (spatial, temporal, GAM) and then computed solutions (and even DOF and GCV when appropriate). However, as the previous description somehow hints, this method performs several conceptually independent operations that might be split for convenience. This work has already been performed by Negri for the FPCA, where the analogous `apply` function in `Mixed_FE_FPCA.h` only covers the "second section" of its homonym and is preceded by a general private setter called `SetAndFixParameters` that covers matricial computation. We decided to perform the same conceptual division in regression, by creating a new function called `preapply` performing similar tasks - that we report in Appendix B - still keeping the old `apply` name for the latter part.

Remark 1. Note that the version in which the functions appear in the appendix is the final one, also comprehensive of the modifications introduced for the sake of our λ optimization part. Moreover full understanding of these terms is indissolubly subject to a precise knowledge of the class structure, we just leave this reference for expert users or future developers that might be interested in seeing the state of the art after our work. A novice reader might continue to follow this Section without mitigating the grasp on the key topics.

Getting back to the partition, our decision roots in two reasons: symmetry and its use for de-templatization. Indeed, as it can be seen in [5], almost all templates used in the original `apply` belong to its "first section". To be more precise these templates are:

- `InputHandler`: unavoidable, used extensively across the class template. It stores the `RegressionData` type and thus the "id" of specific regression problem to be solved. This is the only template parameter that will be kept global for the class.

- `IntegratorSpace`: of the whole template class, its only usage can be found in the "first section" of the old `apply`, in order to assemble (once and for all, since they do not depend from spatial or temporal λ) R_0 and R_1 matrices defined in equation (1.7); nowhere else.
- `ORDER`, `mydim`, `ndim`: used in just two cases: the one described for `IntegratorSpace` and to store the `mesh` inside the class. It is important to notice that this storage is easily avoidable since the true use of the mesh is reduced just to a couple of situations: the evaluation of the basis function matrix Ψ (see equation (1.6)) in `void setPsi(void)` and - for areal data - also for `void setA(void)`, the setter of the areal matrix A defined in 1.4.1. Both functions are called just once in the initial lines of the old `apply`. There is margin for improvement, we will return on this later.
- `IntegratorTime`, `SPLINE_DEGREE`, `ORDER_DERIVATIVE`: like in the previous case, these template parameters are used just once in the class: at the end of the first part of the old `apply` in order to call the universal temporal matrices builder employed just for space-time problems: `void buildSpaceTimeMatrices(void)`.

In light of these remarks, we decided to perform the following template rationalization in class `Mixed_FE_Regression.h`:

1. Let both `preapply` and `apply` public members to be called - as was just the `apply` in [5] - by `regression_skeleton` in `Regression_Skeleton.h` (a level where the mesh is constructed and thus available).
2. Remove all template parameters from the class but the essential `InputHandler`. In this way it becomes vital to pass the mesh as parameter to the `preapply` early builder that hence assumes the form of a template function:

```
template<UInt ORDER, UInt mydim, UInt ndim, typename IntegratorSpace, typename
    IntegratorTime, UInt SPLINE_DEGREE, UInt ORDER_DERIVATIVE, typename A>
  void preapply(EOExpr<A> oper,const ForcingTerm & u, const MeshHandler<ORDER,
      mydim, ndim> & mesh_ )
```

3. Make functions `setA` and `setPsi` mesh dependent and thus template functions.

```
template<UInt ORDER, UInt mydim, UInt ndim>
   void setPsi(const MeshHandler<ORDER, mydim, ndim> & mesh_);
template<UInt ORDER, UInt mydim, UInt ndim>
  void setA(const MeshHandler<ORDER, mydim, ndim> & mesh_);
```

4. Store in the class the space-time mesh number of nodes that is a template independent parameter used in several private function members.

```
const UInt N_;   // Number of spatial basis functions.
const UInt M_;   // Number of temporal nodes
```

5. Make `void buildSpaceTimeMatrices(void)` a template function dependent on the temporal parameters `IntegratorTime`, `SPLINE_DEGREE`, `ORDER_DERIVATIVE`:

```
template<typename IntegratorTime, UInt SPLINE_DEGREE, UInt ORDER_DERIVATIVE>
  void buildSpaceTimeMatrices(void);
```

Note: The same operations have been performed "in small" for template classes contained in `Mixed_FE_FPCA.h`. Of course, in that case we only had to remove spatial parameters. We will avoid to devote an entire Section to the details, since these topics are a bit far from the core of our project, moreover the simple modifications can be easily derived by analogy. For further insight on the subject, please refer directly to the code.

2.5 A matter of symmetry

During the analysis of the previous versions of the library we noticed a theoretical error in the computation of the exact GCV function. In particular, we noticed that the solver used for the inversion necessary in the computation of the matrix S (see (3.3)) was `Eigen::LDLT`. This solver follows the decomposition as stated in the following theorem (see, e.g., [16]).

Theorem 2.5.1. Let B be a symmetric and positive (respectively, negative) definite $n \times n$ matrix, $n \in \mathbb{N}$. Then it holds:

$$B = LDL^T,$$

where L is a lower triangular matrix and D is a diagonal matrix.

Now, analyzing the structure of $S = \Psi(\Psi^T A Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} \Psi^T A Q$, we observe that the matrix to be inverted is

$$T = \Psi^T A Q \Psi + \lambda R_1^T R_0^{-1} R_1.$$

Clearly, the second term in the summation is symmetric, being R_0 the mass matrix of the system, but the first addendum in general could be not symmetric. In particular, if we do not have areal data, i.e., A, the incidence matrix, is the identity, we obtain

$$T = \Psi^T Q \Psi + \lambda R_1^T R_0^{-1} R_1.$$

and the first addendum is still symmetric, since the matrix Q is a projection matrix, symmetric by construction ($Q^T = Q$).
Nevertheless, when we are dealing with areal data, the transpose of the matrix $\Psi^T A Q \Psi$ is

$$(\Psi^T A Q \Psi)^T = \Psi^T Q^T A^T \Psi = \Psi^T Q A^T \Psi$$

where, in the last equality, we exploited the symmetry of Q. We clearly see that in general $QA^T \neq AQ$ from this trivial counterexample: let

$$Q = \begin{bmatrix} 2 & 3 \\ 3 & 2 \end{bmatrix}$$

and

$$A = \begin{bmatrix} 6 & 0 \\ 0 & 1 \end{bmatrix}$$

then $AQ = \begin{bmatrix} 12 & 18 \\ 3 & 2 \end{bmatrix} \neq \begin{bmatrix} 12 & 3 \\ 18 & 2 \end{bmatrix} = QA^T$.
Then in this case the matrix T is non-symmetric, and the solver `LDLT` is not usable. The alternative to this solver in Eigen is the `PartialPivLU` solver: according to the performances, even though it is a bit more accurate than the other one, it is slower. Thus, we decided to keep the `LDLT` solver whenever the regression is not of areal type, whereas we use `PartialPivLU` only in the areal case. This is made possible by the `std::conditional` structure:

```
1 typedef typename std::conditional<std::is_base_of<Areal, InputCarrier>::value,
    Eigen::PartialPivLU<MatrixXr>, Eigen::LDLT<MatrixXr>>::type Factorizer;
2 Factorizer factorized_T(T);
```

The variable `Factorizer` then becomes of type `Eigen::PartialPivLU<MatrixXr>` for areal data, and of type `Eigen::LDLT<MatrixXr>>` in all the other cases (see next Sections for the description of the template parameters `Areal` and `InputCarrier`). We recall that `MatrixXr` is the name for the Eigen matrix `Eigen::Matrix<Real,Eigen::Dynamic,Eigen::Dynamic>` (defined via `typedef` in the file `src/Fda_PDE.h`).
In this way we do not lose in performance, but we correct at the same time the error which was present in the previous version of the library.

Chapter 3

In-depth Analysis of GCV

This theoretical Chapter describes in detail our study of Generalized Cross Validation (GCV) and explores possible optimization methods to find its extremes.

Before our project, the only smoothing parameter selection criterion available was the grid evaluation, which is not efficient for many reasons. First of all, the grid must be given as input by the user, thus it could be chosen in a completely wrong way, far from the optimal value, by a naive user, leading to completely wrong estimations. Indeed, the smoothing parameter is crucial to have a correct adherence to the data and obtain good estimates. Moreover, also an expert user should repeat the choice of the grid many times, each of them centering and restricting the grid around the guessed optimal value, in order to be as close as possible to it, thus losing time in calling the smoothing function repeatedly.

The optimization methods we propose are instead automatic and - without any input from the user - are able to find in most of the cases an optimal value of the smoothing parameter λ, with only one call to the smoothing function. Furthermore, we stress again the fact that the actual default choice, which is the stochastic grid evaluation of GCV, is not robust, since in many non pathological cases it leads to completely wrong choices of the smoothing parameters, giving misleading results in the smoothing procedure, as we will show in the test section 5. On the other hand, we will point out that our new optimization methods are more robust and reach a reasonably optimal value also in those cases when the stochastic grid evaluation of GCV fails.

Therefore we need now to define and deeply analyze the GCV, computing its first and second derivatives, which are at the basis of the Newton's optimization methods. Given the definitions of GCV function described in [4], in Section 3.1 we derive the the expression of its first two derivatives as functions of the uni-dimensional smoothing parameter λ, paying attention to the computational burden of the single steps with an eye projected onto the implementation that will be described in chapter 4. In Section 3.2 we describe how to apply Newton's method to the optimization, even with the aid of finite differences in order to achieve the optimal solution also in the cases of absence of an analytic expression of the derivatives (e.g., in the stochastic GCV computation).

3.1 Generalized Cross Validation criterion

In the following analysis, we consider the most general case: with areal data, presence of forcing term, homogeneous Neumann conditions (for Dirichlet boundary conditions see, e.g. [1]: in the code we did not modify the computations in this case): the less general cases can be easily obtained by setting $A = I$ (pointwise data), $Q = I$ (no covariates) or $\mathbf{u} = \mathbf{0}$, according to the necessities. Let $\hat{\mathbf{z}} = W\hat{\boldsymbol{\beta}} + \Psi\hat{\mathbf{f}} = H\mathbf{z} + Q\Psi\hat{\mathbf{f}}$. Then, we obtain from the regression system that

$$\mathbf{f} = (\Psi^T AQ\Psi + \lambda R_1^T R_0^{-1} R_1)^{-1}(\Psi^T AQ\mathbf{z} + \lambda R_1^T R_0^{-1}\mathbf{u}). \tag{3.1}$$

Therefore $\hat{\mathbf{z}}$ can be expressed as

$$\hat{\mathbf{z}} = (H + QS(\lambda))\mathbf{z} + \mathbf{r}(\lambda), \tag{3.2}$$

where

$$\begin{cases} S(\lambda) = \Psi(\Psi^T A \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} \Psi^T A Q \\ \mathbf{r}(\lambda) = \lambda Q \Psi (\Psi^T A Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} R_1^T R_0^{-1} \mathbf{u} \end{cases} \tag{3.3}$$

and then the GCV function we are going to study is

$$GCV(\lambda) = \frac{n}{(n - tr(H + QS(\lambda)))^2} ||\mathbf{z} - \hat{\mathbf{z}}||_2^2 = \frac{n}{(n - (k + tr(S(\lambda))))^2} ||\mathbf{z} - \hat{\mathbf{z}}||_2^2, \tag{3.4}$$

where $||\cdot||_2$ is the Euclidean norm and $tr(H + QS(\lambda))$ is the number of Degrees Of Freedom, dof. We now show the computation of the first and second derivatives, which were not considered in the previous versions of the code. We have organized the computations in order to be the most efficient as possible, saving some reused parts of computations.

3.1.1 First derivative of GCV function

First of all, we recall the fundamental lemma of calculus for the derivative of the inverse of a non-singular matrix depending on a (real) parameter:

Lemma 3.1.1. Let $B = B(\lambda) \in \mathbb{R}^{n \times n}$, $n \in \mathbb{N}$: we have

$$\frac{\partial (B^{-1})}{\partial \lambda} = -B^{-1} \frac{\partial B}{\partial \lambda} B^{-1} \tag{3.5}$$

Proof. We have $BB^{-1} = I$: then, we apply Leibniz rule to get $\frac{\partial B}{\partial \lambda} B^{-1} + B \frac{\partial (B^{-1})}{\partial \lambda} = 0$ and the assertion follows. □

We can compute the following matrices and derivatives:

1. $R = R_1^T R_0^{-1} R_1$
2. $T = \Psi^T A Q \Psi +_1^T R_0^{-1} R_1 = \Psi^T A Q \Psi + \lambda R$
3. $V = (\Psi^T A Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} \Psi^T A Q = T^{-1} \Psi^T A Q$
4. $K = T^{-1} R$
5. $F = KV$
6. $S = \Psi(\Psi^T A Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} \Psi^T A Q = \Psi V$
7. $\frac{dS}{d\lambda}$
   $= -\Psi(\Psi^T A Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} R_1^T R_0^{-1} R_1 (\Psi^T A Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} \Psi^T A Q$
   $= -\Psi T^{-1} R V = -\Psi K V = -\Psi F.$

We then introduce some auxiliary variables:

$$\hat{\varepsilon} = \mathbf{z} - \hat{\mathbf{z}} = (Q - QS(\lambda))\mathbf{z} - \mathbf{r}(\lambda) \tag{3.6}$$

$$SS_{res} = < \hat{\varepsilon}, \hat{\varepsilon} > \tag{3.7}$$

$$dor = n - k - tr(S(\lambda)) = n - dof \tag{3.8}$$

and we define the following quantities:

1. $\mathbf{f} = R_1^T R_0^{-1} \mathbf{u}$
2. $\mathbf{g} = T^{-1} \mathbf{f}$

3. $\mathbf{t} = \frac{dS}{d\lambda}\mathbf{z}$
4. $\mathbf{h} = (\lambda K - I)\mathbf{g}$
5. $\mathbf{s} = \lambda\Psi T^{-1}R_1^T R_0^{-1}\mathbf{u} = \lambda\Psi\mathbf{g}$
6. $\mathbf{r}(\lambda) = \lambda Q\Psi T^{-1}R_1^T R_0^{-1}\mathbf{u} = \lambda Q\Psi\mathbf{g} = Q\mathbf{s}$
7. $\frac{d\mathbf{s}}{d\lambda} = \Psi\mathbf{g} - \lambda\Psi T^{-1}RT^{-1}R_1^T R_0^{-1}\mathbf{u} = \Psi\mathbf{g} - \lambda\Psi K\mathbf{g} = \Psi(I - \lambda K)\mathbf{g}$
8. $\mathbf{p} = -(\frac{dS}{d\lambda}\mathbf{z} + \frac{d\mathbf{s}}{d\lambda}) = \Psi\mathbf{h} - \mathbf{t}$
9. $a =< \mathbf{p}, \hat{\boldsymbol{\varepsilon}} >$.

We can thus compute the first derivative of the GCV function exploiting the notation just introduced, remembering the properties of the orthogonal projector $\mathbf{Q}$, i.e. $\mathbf{Q}^2 = \mathbf{Q}$:

$$\frac{dGCV}{d\lambda}(\lambda) = \frac{d(n\frac{SS_{res}(\lambda)}{dor^2(\lambda)})}{d\lambda} = n(\frac{dSS_{res}}{d\lambda}\frac{1}{dor^2} + SS_{res}\frac{d(dor^{-2})}{d\lambda}) = n(A\cdot\frac{1}{dor^2} + SS_{res}\cdot B)$$

where:

$$A = \frac{dSS_{res}}{d\lambda} = \frac{d(<\hat{\boldsymbol{\varepsilon}},\hat{\boldsymbol{\varepsilon}}>)}{d\lambda} = 2 < \frac{d\hat{\boldsymbol{\varepsilon}}}{d\lambda}, \hat{\boldsymbol{\varepsilon}} >= 2 < \frac{d((Q-QS)\mathbf{z}-\mathbf{r}(\lambda))}{d\lambda}, \hat{\boldsymbol{\varepsilon}} >= -2 < Q(\frac{dS}{d\lambda}\mathbf{z} + \frac{d\mathbf{s}}{d\lambda}), \hat{\boldsymbol{\varepsilon}} >$$
$$\overset{Q\in Sym}{=} -2 < \frac{dS}{d\lambda}\mathbf{z} + \frac{d\mathbf{s}}{d\lambda}, Q\hat{\boldsymbol{\varepsilon}} > \overset{Q^2=Q}{=} 2 < \Psi\mathbf{h} - \mathbf{t}, \hat{\boldsymbol{\varepsilon}} >= 2a$$

$$B = \frac{d(dor^{-2})}{d\lambda} = -2\frac{1}{dor^3}\frac{d(dor)}{d\lambda} = -2\frac{1}{dor^3}\frac{d(n-k-tr(\mathbf{S}))}{d\lambda} = 2\frac{1}{dor^3}tr(\frac{dS}{d\lambda}).$$

Thus we obtain the desired derivative:

$$\frac{dGCV}{d\lambda}(\lambda) = n(\frac{2a}{dor^2} + SS_{res}\frac{2}{dor^3}tr(\frac{dS}{d\lambda})) = \frac{2n}{dor^2}(\hat{\sigma}^2 tr(\frac{dS}{d\lambda}) + a) \tag{3.9}$$

where we have defined the squared of the standard deviation as

$$\hat{\sigma}^2 = \frac{SS_{res}}{dor}.$$

3.1.2 Second derivative of GCV function

To compute the second derivative of the GCV function in an effective way, we exploit the same values and notations previously introduced, adding the following definitions:

1. $\frac{d^2S}{d\lambda^2}$
   $= 2\Psi[(\Psi^T A Q\Psi + \lambda R_1^T R_0^{-1}R_1)^{-1}R_1^T R_0^{-1}R_1]^2(\Psi^T A Q\Psi + \lambda R_1^T R_0^{-1}R_1)^{-1}\Psi^T AQ$
   $= 2\Psi(T^{-1}R)^2 V = 2\Psi K^2 V = 2\Psi KF$
2. $\frac{d\mathbf{h}}{d\lambda} = -2K\mathbf{h}$
3. $b =< \mathbf{p}, Q\mathbf{p} >$
4. $c =< \frac{d\mathbf{p}}{d\lambda}, \hat{\boldsymbol{\varepsilon}} >$

and then we get:

$$\frac{d^2GCV}{d\lambda^2}(\lambda) = 2n\frac{d(dor^{-2}(\hat{\sigma}^2 tr(\frac{dS}{d\lambda}) + < \mathbf{p}, \hat{\boldsymbol{\varepsilon}} >))}{d\lambda} = 2n(C + D + E + F + G). \tag{3.10}$$

We can, indeed, divide the computations as follows:

$C = \frac{d(dor^{-2})}{d\lambda}(\hat{\sigma}^2 tr(\frac{dS}{d\lambda}) + < \boldsymbol{p}, \hat{\boldsymbol{\varepsilon}} >) = 2\frac{1}{dor^3} tr(\frac{dS}{d\lambda})(\hat{\sigma}^2 tr(\frac{dS}{d\lambda}) + < \boldsymbol{p}, \hat{\boldsymbol{\varepsilon}} >)$

$D = \frac{1}{dor^2}\frac{d(\hat{\sigma}^2)}{d\lambda} tr(\frac{dS}{d\lambda}) = \frac{1}{dor^2}\frac{\frac{d(SS_{res})}{d\lambda} dor - SS_{res}\frac{d(dor)}{d\lambda}}{dor^2} tr(\frac{dS}{d\lambda}) = \frac{1}{dor^2}\frac{2<\mathbf{p},\hat{\boldsymbol{\varepsilon}}> dor + SS_{res} tr(\frac{dS}{d\lambda})}{dor^2} tr(\frac{dS}{d\lambda})$
$= \frac{1}{dor^3}(2 < \mathbf{p}, \hat{\boldsymbol{\varepsilon}} > + \hat{\sigma}^2 tr(\frac{dS}{d\lambda}))tr(\frac{dS}{d\lambda})$

$E = \frac{1}{dor^2}\hat{\sigma}^2 tr(\frac{d^2S}{d\lambda^2})$

$F = \frac{1}{dor^2} < \frac{d\mathbf{p}}{d\lambda}, \hat{\boldsymbol{\varepsilon}} >= \frac{1}{dor^2} < -\frac{d^2\mathbf{S}}{d\lambda^2}\mathbf{z} - \frac{d^2\mathbf{s}}{d\lambda^2}, \hat{\boldsymbol{\varepsilon}} >= \frac{1}{dor^2} < -\frac{d^2S}{d\lambda^2}\mathbf{z} - 2\Psi K\mathbf{h}, \hat{\boldsymbol{\varepsilon}} >$

$G = \frac{1}{dor^2} < \mathbf{p}, \frac{d\hat{\boldsymbol{\varepsilon}}}{d\lambda} >= \frac{1}{dor^2} < \mathbf{p}, Q\mathbf{p} >.$

And then we conclude that

$$\begin{aligned}
&\frac{d^2GCV}{d\lambda^2}(\lambda) \\
&= \frac{2n}{dor^2}\{\frac{1}{dor}[3\hat{\sigma}^2 tr(\frac{dS}{d\lambda}) + 4 < \mathbf{p}, \hat{\boldsymbol{\varepsilon}} >]tr(\frac{dS}{d\lambda}) + \hat{\sigma}^2 tr(\frac{d^2S}{d\lambda^2}) + < \mathbf{p}, Q\mathbf{p} > + < \frac{d\mathbf{p}}{d\lambda}, \hat{\boldsymbol{\varepsilon}} >\} \\
&= \frac{2n}{dor^2}\{\frac{1}{dor}[3\hat{\sigma}^2 tr(\frac{dS}{d\lambda}) + 4a]tr(\frac{dS}{d\lambda}) + \hat{\sigma}^2 tr(\frac{d^2S}{d\lambda^2}) + b + c\}. \qquad (3.11)
\end{aligned}$$

In the implementation of the first and second derivative we will use the formulas (3.9) and (3.11), which are effective from the point of view of the minimum number of operations. Nevertheless we recall that these computations are costly due to the inversion of several matrices; we will also show a method of optimization which avoids them.

3.1.3 Stochastic GCV

Another criterion to find the smoothing parameter λ is the Stochastic Generalized Cross Validation. In this case we describe the only function evaluation, since it is based on a statistical procedure and the computation of the derivatives is not affordable. The optimization method we will implement for this function will not use the derivatives, indeed. To introduce the function, we start presenting the following result:

Lemma 3.1.2. Let S be a symmetric matrix and let $\mathbf{u} = (u_1, \cdots, u_N)^T$ a vector of N independent samples from a random variable U, which takes values -1 and 1 each with probability 0.5. Then

$$\mathbb{E}[\mathbf{u}^T S\mathbf{u}] = tr(S).$$

Applying this estimator, the computation of the Degrees Of Freedom *dof* becomes:

$$dof = k + tr(S) = k + [\mathbf{u}^T\Psi T^{-1}\Psi^T AQ\mathbf{u}]$$

Then, to approximate the mean value, a chosen number of realization of $\mathbf{u}$ are fixed, in general 100, and the sample mean of the argument of $\mathbb{E}$ is computed. The computation of the GCV then goes exactly as in the case of the exact GCV:

$$GCV(\lambda) = \frac{n}{(n - (k + tr(S(\lambda))))^2}||\mathbf{z} - \hat{\mathbf{z}}||_2^2,$$

This method is much more efficient than the computation of the exact GCV, as we will see in the testing section (see also [3]).

3.2 The Newton's method

In order to find the optimal value of the GCV function we chose to exploit Newton's method: in particular we need to find the root of the derivative of the function. In general the GCV function has a particular shape which guarantees a single minimum, thus finding the zero of the derivative of the function seems to be the right choice. Let $f : \mathbb{R} \to \mathbb{R}$ a generic function, which could be the GCV function or another one, supposed to be sufficiently regular (at least $C^3(\mathbb{R})$), and let $\alpha \in \mathbb{R}$ the single zero, such that $f''(\alpha) \neq 0$ (notice that the GCV in general respects this hypothesis, because $f''(\alpha) > 0$, since it is convex in a neighborhood of the minimizer α). Then Newton method reads (see [11]):
up to convergence, fixed an initial guess x_0, for every $n > 0$ compute

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

where f' and f'' are respectively the first and second derivatives if f. As stopping criterion we compute the residual and compared with TOL, stopping the iterations if

$$|f'(x_n)| \leq \text{TOL}$$

We decided to leave the possibility to choose the parameter TOL to the user (called `stop_criterion_tol` in the `R` function), but we nevertheless defaulted the value to 0.05, which seemed a reasonable tolerance to have good results. The Newton's method has a quadratic order of convergence (if the initial guess is sufficiently close to the real root) to the zero α, meaning that

$$|x_{n+1} - \alpha| \leq C|x_n - \alpha|^2 \quad \forall n \geq 0$$

As we can see, Newton's method is computationally demanding, because we need to compute both the first and the second derivatives of the GCV function, which could be inefficient due to the solution of very big linear systems involved in the computation. Moreover, this method cannot be used if we do not know the analytical expression of the derivatives of the function, which is the case, for example, of the stochastic GCV.
Then, we decided to propose another method, derived from the one just presented and then compared in efficiency with it. In particular, we substitute the exact derivatives with approximated derivatives computed by means of centered finite differences (see again [11]), namely we approximate them in the following way, having set $h = 4\text{e}{-06}$:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} =: f'_{CD}(x) \tag{3.12}$$

and

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} =: f''_{CD}(x) \tag{3.13}$$

we know, by Taylor expansion with Lagrange form of the remainder, that

$$f'(x) = f'_{CD}(x) + \mathcal{O}(h^2) \tag{3.14}$$

and

$$f''(x) = f''_{CD}(x) + \mathcal{O}(h^2) \tag{3.15}$$

The method thus reads:
up to convergence, fixed an initial guess x_0, for every $n > 0$ compute

$$x_{n+1} = x_n - \frac{h}{2} \frac{f(x_n + h) - f(x_n - h)}{f(x_n + h) - 2f(x_n) + f(x_n - h)} \tag{3.16}$$

The underlying idea of the method can be explained with the following two-step procedure:

Step 1

We approximate f' with f'_{CD} defined in (3.12) and look for the zero of f'_{CD}. Indeed, if $\tilde{\alpha}$ is a zero of f'_{CD}, then

$$|f'(\tilde{\alpha})| = |f'(\tilde{\alpha}) - f'_{CD}(\tilde{\alpha})| = \mathcal{O}(h^2),$$

meaning that if we chose h small enough, we can say that $f(\tilde{\alpha})$ is a good approximation of the minimum value of the function f, being the derivative $f'(\tilde{\alpha})$ very small in modulus. We are then led to find the root $\tilde{\alpha}$ of f'_{CD}.

Step 2

The Newton's method applied to f'_{CD} should read:
up to convergence, fixed an initial guess x_0, for every $n > 0$ compute

$$x_{n+1} = x_n - \frac{f'_{CD}(x_n)}{\frac{d}{dx}f'_{CD}(x_n)} \tag{3.17}$$

but we recall that, due to (3.14), we have also

$$f''(x) = \frac{d}{dx}f'_{CD}(x) + o(1) \tag{3.18}$$

which implies, together with (3.15):

$$\frac{d}{dx}f'_{CD}(x) = f''_{CD}(x) + o(1) \tag{3.19}$$

substituting (3.19) in (3.17), we obtain the following method, which is exactly the method (3.16):
up to convergence, fixed an initial guess x_0, for every $n > 0$ compute

$$x_{n+1} = x_n - \frac{f'_{CD}(x_n)}{f''_{CD}(x_n)} \tag{3.20}$$

This strategy is still convergent (if the initial guess is sufficiently close to the real root) in the case of GCV function, as we will see in Chapter 5, dedicated to the tests.
Actually, due to the particular shape of the GCV function, we have found that in many cases the value of $\frac{d}{dx}f'_{CD}(\tilde{\alpha})$ is very big, more than $1e7$, then the value of h chosen is very small compared to it, and thus the effectiveness of the method is enforced. As we have seen, from step 2 we obtain the convergence of the method to a root of f'_{CD}, which is from step 1 very close to the real root of f', which is our target.

In conclusion, we recall that the importance of this approximated method is also that it can be used in the cases when we do not have a precise analytical expression of the function, but we can only compute its value. For example, in the case of the stochastic GCV we only have the evaluation of the function, and we cannot easily compute the derivatives, thus in this case we will be forced to use only the finite differences approximation. Also in the case of and in the case of Dirichlet boundary conditions, since the matrices are modified in some of the entries, we do not have an analytical expression for the GCV derivatives.

3.2.1 Initialization of Newton's method

For what concerns the initial guess for the Newton's method, we built an automatic way to find a generally good initial condition (generally in the sense that the GCV function can have strange shapes in some particular cases: in presence of more than one local minimum, for example, the optimization method could find not the best minimum). To initialize the method, we decided to evaluate six points from 5e−05 to 1e+03, which are distributed in a uniform logarithmic way, i.e. they are equispaced in on a logarithmic axis: values are collected in the following vector:

$vals = [5.00000\text{e}{-05},\ 1.442700\text{e}{-03},\ 4.162766\text{e}{-02},\ 1.201124,\ 3.465724\text{e}01, 1\text{e}{+03}]^T$.

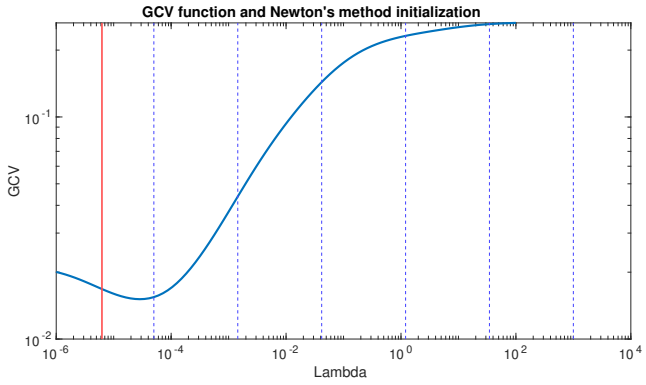
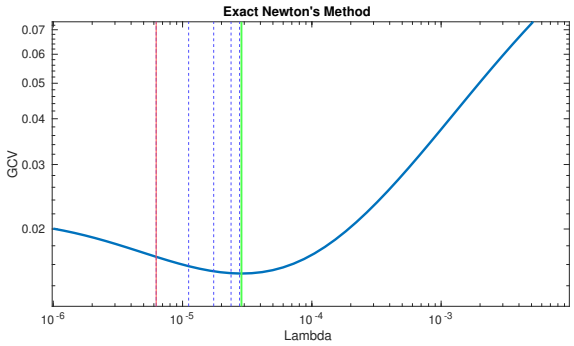


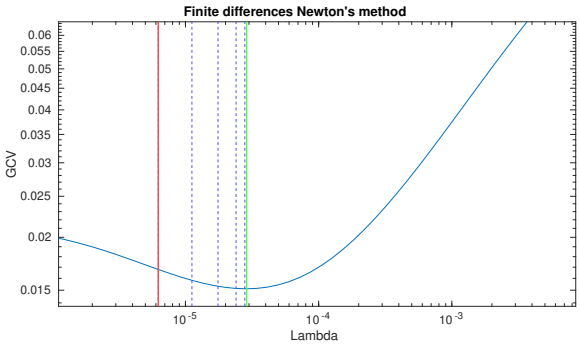
Figure 3.1: GCV function in log-log plot.
Dashed blue lines - explored initial values, red line - automatically chosen initial λ_0.

We compute the GCV function over each of these values and then we initialize the method with the λ which gives the minimum GCV evaluation. Namely, we start form $\lambda_0 = \alpha\lambda$, in order to get more stability: we found, with different numerical tests, that $\alpha = 0.125$ is a good safety parameter to obtain a good convergence of the method. Indeed, we have noticed that reducing the found initial value of a such coefficient gives better results in terms of number of iterations needed to find the minimum up to the prescribed tolerance TOL. In Figure 3.1 we show the GCV function for a test case, which will be described in the following Sections (Section 5.1.1, case with 1'681 observation locations, coinciding with the nodes). We can observe that the initialization is in the region where the function is convex, thus it will lead to the identification of the correct minimum up to the prescribed tolerance.

We can then show, for the same test case, with $TOL = 0.05$, the iterative procedure of the two optimization methods described: in Figure 3.2a we can see the optimization obtained by means of exact Newton's method, whereas in Figure 3.2b we see the optimization procedure for the finite differences Newton's method: we can see that in the same number of iterations, 6, they reach similar values, 2.854987e−05 for the exact Newton's method and 2.879755e−05 for the finite differences method (the relative difference is 0.88%, almost negligible). This value is also very close to the minimum of the function, as we can see from the same plots.



(a) Newton's exact iterative procedure.



(b) Newton's finite differences iterative procedure.

Figure 3.2: The minimum value is reached in 6 iterations with both methods.
Red line - initial λ_0, dashed lines - intermediate steps, green line - optimal value.

Chapter 4

Lambda Optimization

Besides code reorganization - described in Chapter 2 - and some minor error fixing, our work has been mainly devoted to a pioneering goal: finding a way to introduce optimization methods in the code, without totally disrupting the original regression flow described in `Mixed_FE_Regression.h`. The work has been performed under a limitation, i.e. we had to implement optimization under the pure spatial regression problem. However, we also had to face three major challenges:

1. **Portability among structures**: as reminded in the introduction we started working on `fdaPDE` library using the version proposed by Negri [9], in which just spatial regression was present. Having our effort spanned more than a year, we had constantly to keep track of the modifications and evolutions the library was undergoing, regularly updating our work to the latest version. We soon started to understand how nonsensical was treating our project in a pure standard way. Trying massive modifications in file `Mixed_FE_Regression.h` - as done e.g. by Ardenghi-Vicini [1] or Colombo-Perin [5] - immediately proved its infeasibility, at least for two reasons. Impossibility to continually keep our structure updated while the files were modified by almost every project concerning regression, not to mention the risk of finding sooner of later some major incompatibility issue between our ideas and the work of someone else.

   Nevertheless, the intrinsic link between the permanently updating sub-structure and our work could not be completely severed to produce a totally independent structure - as done e.g. by Ghilotti-Pigolotti [7], being working on a totally blank background. For this reason, we started to develop our project as a true plugin-unplug structure to be added and removed along the different versions in a small amount of time, with fewer changes to the work of others and keeping itself as a self-contained as possible in terms of syntax and usage. We will detail more this effort when talking about the `Carrier` class variadic template in Section 4.2, introduced mainly for this purpose.

2. **Extensibility to other methods**: a second issue we had to address was the necessity to create code being as general as possible, conceived to be effective on spatial problems, but also able to be generalized - when necessary - to spatio-temporal regression or even GAM. It's interesting to stress that all the classes we built are particularizations for unidimensional lambda optimization, but have already been conceived with the intent of being extensible to bivariate problems (like separable spatio-temporal regression) or even - to some extent - FPCA. This imprint was taken from the beginning, even without projects regarding these arguments being completed. However we can confirm, *a posteriori* that everything we wrote is fully able to be generalized for the methods having been written so far, obviously with a bit of effort. In this way, we feel like having been able to accomplish the role of pioneers we mentioned at the beginning, clearing a path for a full definition of optimized methods for the whole regression-based part of the library (note that - in particular - we have already prepared the full `R` syntax for this time to come).

3. **Efficiency** Goes without saying that for an optimization problem efficiency is a parameter of paramount importance. However, its meaning can be declined in several ways: first of all we tried to build mathematically efficient optimization methods by making use of the (quadratic speed) Newton algorithms and stochastic GCV computation (whose efficiency is proven e.g. in [3]). In these cases we tried to compare time and accuracy of our calculations with the benchmark grid method present in the current version of the library - now adjusted to fit consistently in our new superstructure -, showing when and to which extent we can assume to have brought an improvement. Analogously, we kept an eye on efficiency from the standpoint of the aforementioned `Carrier`, making sure its presence would not affect the global performance by addition of useless copies (for this reason we will introduce an *ad hoc* pointer based syntax). Finally we also strove to identify a structure for the lambda optimizer capable of making the richest output possible with the smallest effort allowable, in particular minimizing the updates necessary for the computation of the quantities of interest, still keeping into account elegance and generality of the structure, favouring whenever possible a compile time effort rather than a running time overhead.

4.1 Optimization data transmission and storage

4.1.1 `OptimizationData`

The first step we took in order to define our lambda optimization procedure was the definition, at `C++` level, of a true class for collecting all the information regarding data related to optimization. Version Colombo-Perin [5] of the code, but also the previous ones, have never managed these kind of inputs in an independent way, conversely they have always kept them primarily inside class `RegressioData` in header `Regression_Data.h`, probably for the sake of simplicity or because separate management never proved more appropriate. We decided, led by generality, but also for direct employment in our structures, to separate them in a different class, called for consistency `OptimizationData`, to be put in a new header (ad associated source) `Optimization_Data.h`.

This class, as the name suggests, is meant to collect all optimization related information both for grid evaluation, and for Newton based lambda optimization that we will discuss in more detail afterwards. Please, recall that the "grid" is simply the way we call the redesigned grid of λ - or (λ_S, λ_T) in case of spatio-temporal separable problems - passed by the user on which to evaluate the results and possibly select the "best value", i.e. the method employed in the library since its birth.

Concerning the content of the class, the private members are described in Table 4.1, while a brief comment on their purpose can be found in Table 4.2; note that each one is endowed its own getter and setter. Just a few clarifications: many of the members seem to be initialized with non consistent values. Actually this is the case. However, it is irrelevant from a practical viewpoint, since the class is built through two possible constructors (one for pure spatial and GAM problems, one for spatio-temporal regression), that provide meaningful values for all terms needed by the specific problem. Precisely these "consistent" default values come from user defined `R` parameters or `R`-code defaults (contained in the user-interface regression `R` functions `smooth.FEM` [SF] and `smooth.FEM.time` [SFT]). We want to emphasize this is not something we have introduced, but rather a common practice for several classes of the library, e.g. check FPCA code for an example. Thus, for symmetry reasons, we have kept this convention, while using specific `C++`-level defaults for debugging purpose. Therefore, we highly suggest future workers on the library to modify as little as possible the `R` defaults, because they are the meaningful ones!

Type	Name	Default	Admissible
string	criterion	"grid"	"grid" "newton" "newton_fd"
string	DOF_evaluation	"not_required"	"not_required" "stochastic" "exact"
string	loss_function	"unused"	"unused" "gcv"
vector < Real >	lambda_S	$\{-1.0\}$	> 0
vector < Real >	lambda_T	$\{-1.0\}$	> 0
UInt	size_S	1	≥ 1
UInt	size_T	1	≥ 1
UInt	best_lambda_S	0	≥ 1
UInt	best_lambda_T	0	≥ 1
Real	best_value	max()	> 0
Real	initial_lambda_S	0.0	> 0
Real	initial_lambda_T	0.0	> 0
UInt	seed	0	≥ 1
UInt	nrealizations	100	≥ 1
Real	last_lS_used	infinity()	> 0
Real	last_lT_used	infinity()	> 0
Real	current_lambdaS	-1.0	> 0
MatrixXr	DOF_matrix		> 0
Real	tuning	1.0	> 0
Real	stopping_criterion_tol	0.05	$(0, 1)$
vector < Real >	lambdaS_backup		> 0

Table 4.1: Content of `OptimizationData` [`UInt` = `int`, `Real` = `double`, `MatrixXr` see Section 2.5]

Note: very important exception to the previous statement. The default size of `lambda_*` ($* = \mathtt{T}, \mathtt{S}$) vectors *must* be ≥ 1, for how the `Mixed_FE_Regression`'s `apply` method has been designed. I.e, using a true/fake cycle on these vectors depending on the type of regression employed. Present in Colombo-Perin [5], we decided to keep this structure in our version with minor modifications for consistency reasons, see Appendix B for further detail. Hence, even if containing non meaningful terms, *never* let non-defaulted these vectors, or better, never let their length 0.

4.1.2 R interface

In this Subsection we would also like to discuss the main modifications employed in `R` code to prepare a structure able to fit the changes at `C++` level. We can summarize the effort in two phases: adjustment of input parameters with integrated update of consistency checks and output design. We will discuss the latter part at the end of this Chapter in Section 4.7, let's start from the first half. In order to define the variations with reference to [5], we will show as an example `R` user-interface function `smooth.FEM` [SF] - the `fdaPDE` method for spatial and GAM regression with differential regularization. Similar work has been performed also for the temporal function `smooth.FEM.time` [SFT], but - this not being a core argument of our project - we leave to the interested reader the small effort of deducing by analogy the modified interfaces. It is only important to highlight that, besides the interface change, the sole method still available is the old `grid`. However, to aid future workers, we have already prepared the `R` call for when optimized methods will be available also in temporal problems.

Type	Content	R-caller
`criterion`	Describes the optimization method that will be used to manage the penalization parameter	`lambda.selection.criterion`
`DOF_evaluation`	Defines if DOF have to be computed and in case with which method	`DOF.evaluation`
`loss_function`	Defines if the various penalization parameters have to be compared and in case with which method	`lambda.selection.lossfunction`
`lambda_S`	In space-time separable problems is the vector of spatial penalizations in all the other cases is the (only) vector of parameters. Used for `grid` methods.	`lambda` [SF] or `lambdaS` [SFT]
`lambda_T`	In space-time separable problems is the vector of temporal penalizations no meaning otherwise. Used for `grid` methods.	`lambdaT` [SFT]
`size_S` `size_T`	Size of the corresponding vector of penalization parameters. Used for `grid` methods.	NA
`best_lambda_S` `best_lambda_T`	Index of the best lambda inside the corresponding vector of penalization parameters. Used for `grid` `gcv` methods.	NA
`best_value`	Value of the loss function in correspondence of the optimal lambda. Used for `grid` `gcv` methods.	NA
`initial_lambda_S`	Value passed to initialize an optimization algorithm. Used for `newton_fd`/`newton` methods.	`lambda` [SF] or `lambdaS` [SFT]
`initial_lambda_T`	Value passed to initialize an optimization algorithm. Will be used for `newton_fd`/`newton` methods.	`lambdaT` [SFT]
`seed`	Seed for replicability of stochastic DOF computation. Used in `stochastic` methods.	`DOF.stochastic.seed`
`nrealizations`	Increasing, tunes the accuracy of stochastic DOF computation. Used in `stochastic` methods.	`DOF.stochastic.realizations`
`last_lS_used` `last_lT_used`	Utilities to avoid useless computations	NA
`current_lambdaS`	Utility to avoid useless computations	NA
`DOF_matrix`	Matrix of Degrees Of Freedom can be passed by the user to compute `gcv` without internal computation speeding up the procedure	`DOF.matrix`
`tuning`	`gcv` inflation factor for GAM	`GCV.inflation.factor`
`stopping_criterion_tol`	Stop criterion for non `grid` methods	`lambda.optimization.tolerance`
`lambdaS_backup`	Copy of `lambda_S`	NA

Table 4.2: Members of `OptimizationData`

Starting from the old material, the aforementioned function `smooth.FEM`, present in Colombo-Perin [5], had the following guise:

```
smooth.FEM <- function(locations=NULL, observations, FEMbasis, lambda,
    covariates=NULL, PDE_parameters=NULL, incidence_matrix=NULL, BC=NULL, GCV=FALSE,
    GCVmethod="Stochastic", nrealizations=100, DOF_matrix=NULL, search="tree",
    bary.locations=NULL, family="gaussian", mu0=NULL, scale.param=NULL,
    threshold.FPIRLS=0.0002020, max.steps.FPIRLS=15, GCV.inflation.factor=1,
    areal.data.avg=TRUE)
```

Where the optimization-related parameters were:

- `lambda`: the real vector of lambdas to be used for regression purposes.
- `GCV`: Boolean, to specify if generalized cross-validation had to be computed or not.
- `GCVmethod`: `"Stochastic"` or `"Exact"`, to specify how to perform DOF calculations.
- `DOF_matrix`, `nrealizations`, `GCV.inflation.factor` with the same meaning expressed in Table 4.2.

As can be seen, there are two major issues with this interface: the order of parameters appears a bit confused, since information were probably added from projects just meeting the demands, without an *a priori* conceived organic structure (this however is just an issue of readability for programmers, irrelevant for users, since parameters can be called with arbitrary ordering in `R` functions). Moreover, as already stated in the previous Subsection, the enrichment provided by our project deemed unavoidable a partial reorganization of optimization based parameters, in particular with the removal of the not very flexible `GCVmethod` and `GCV`, in favour of new, more malleable user interfaces. Here we indicate the new reordered and enhanced call:

```
smooth.FEM<-function(locations=NULL, observations, FEMbasis, covariates=NULL,
    PDE_parameters=NULL, BC=NULL, incidence_matrix=NULL, areal.data.avg=TRUE,
    search="tree", bary.locations=NULL, family="gaussian", mu0=NULL,
    scale.param=NULL, threshold.FPIRLS=0.0002020, max.steps.FPIRLS=15,
    lambda.selection.criterion="grid", DOF.evaluation=NULL,
    lambda.selection.lossfunction=NULL, lambda=NULL, DOF.stochastic.realizations=100,
    DOF.stochastic.seed=0, DOF.matrix=NULL, GCV.inflation.factor=1,
    lambda.optimization.tolerance=0.05)
```

where the new optimization parameters cover the last lines of the function, and whose meaning has already been explained with reference to `C++` code in Table 4.2. For the sake of completeness we will also repeat the description here:

- `lambda.selection.criterion`: `'grid'` [default], `'newton'` or `'newton_fd'`, allows to select the optimization method we want to use for parameter λ.
- `DOF.evaluation`: `NULL` [default], `'stochastic'` or `'exact'`, the type of DOF computation to be performed (if required).
- `lambda.selection.lossfunction`: `NULL` [default] or `'GCV'` the type of loss function to use in smoothing parameter optimization.
- `lambda`: `NULL` [default] or an intial number for optimized methods, a vector for grid evaluation.
- `DOF.stochastic.realizations`: [default 100], the old `nrealizations` for stochastic DOF computation, higher the value, better the accuracy, longer computational times.
- `DOF.stochastic.seed`: [default 0], initial seed for stochastic DOF computation (0 means no seed, random).

- `DOF.matrix`: [default `NULL`] the old `DOF_matrix` for grid evaluations where the DOFs have already been computed.
- `GCV.inflation.factor`: [default 1], as before, for tuning DOFs in GCV computation.
- `lambda.optimization.tolerance`: [default 0.05] useful for selecting tolerance in optimized methods.

Note: In order to facilitate comprehension for future workers on `fdaPDE`, we leave here a simple guide to all the consistency checks and warnings that can be raised whenever improper initialization of the optimization parameters is used, to see direct implementation of these warnings, please check in directory `R`, file `checkSmoothingParameters.R` (or `checkSmoothingParameters_time.R`, if interested in temporal regression):

- **Default**. There are two default methods implemented in the function.
  The first is the *explicit default* that can be deduced by the previous list: pure grid evaluation with no GCV or DOF computed (the simplest and cheapest method). However, grid evaluation, needs a non-`NULL` vector of `lambda` to be available, otherwise, an error will be raised with the following content:

```
stop("lambda required for optimization='grid'; now is NULL.")
```

  This makes the explicit default a bit tricky since the user actually needs to specify some vector of lambdas to perform optimization. In order to overcome this difficulty, and given the promising results that will be discussed in Chapter 5, we have also implemented an *implicit default* to be applied if `lambda` = `NULL`: stochastic DOF finite differences Newton's method, the cheapest, robust optimized strategy implemented in our project. As a consequence, if `lambda` = `NULL` a warning will be issued with the following content:

```
warning("the lambda passed is NULL, passing to default optimized methods.")
```

  Example: suppose we have already defined a set of observations called `data` and a `FEMbasis`. If we want to perform the smoothing and we don't select anything, i.e. we write:

```
output <- smooth.FEM(observations=data, FEMbasis=FEMbasis)
```

  then, it will be equivalent to having called the implicit default:

```
output <- smooth.FEM(observations=data, FEMbasis=FEMbasis, lambda=NULL,
    lambda.selection.criterion='newton_fd', DOF.evaluation='stochastic',
    lambda.selection.lossfunction='GCV')
```

  Conversely suppose the user provides a non-`NULL` homonym vector of `lambda`, then call

```
output <- smooth.FEM(observations=data, FEMbasis=FEMbasis, lambda=lambda)
```

  coincides with:

```
output <- smooth.FEM(observations=data, FEMbasis=FEMbasis, lambda=lambda,
    lambda.selection.criterion='grid', DOF.evaluation=NULL,
    lambda.selection.lossfunction=NULL)
```

- **Lambda**. If instead using non-grid optimization, parameter `lambda` assumes the role of initial value for the optimization algorithm. If a vector will be provided instead of a single real, the elements following the first will be discarded, while a warning will be raised:

```
warning("In optimized methods lambda is the initial value, all terms following
    the first will be discarded.")
```

Remember, proposing an initial value might not be safe for convergence reasons, we have already stressed this concept in Section 3.2.1, therefore *never* use initial values in optimized methods, unless you are very sure of the convergence, leave instead the default `NULL` that provides an automatic sensible method for initializing the algorithm correctly.

- **Optimization**. There are four fundamental rules to be kept in mind when using optimized methods:

  1. non-grid evaluations can't have `DOF.evaluation` = `NULL`.
     We solve this issue depending on the case:
     If we pass a `lambda.selection.method='newton'` and `!DOF.evaluation='exact'`, `DOF.evaluation` will be automatically set as `'exact'` (the sole compatible evaluation strategy) and the following warning will be raised:

     ```
     warning("This method needs evaluate DOF in an 'exact' way, selecting
         'DOF.evaluation'='exact'.")
     ```

     Instead, if the method is `lambda.selection.method='newton_fd'` and we have also `DOF.evaluation=NULL` or we start from baseline `lambda.selection.method='grid'`, we add `lambda.selection.lossfunction='GCV'` and still keeping a `NULL` `DOF.matrix`: `DOF.evaluation` will be automatically set as `'stochastic'` (more efficient than the `'exact'` counterpart) and the following warning will be raised:

     ```
     warning("This method needs evaluate DOF, selecting
         'DOF.evaluation'='stochastic'.")
     ```

     In particular DOF cannot be substituted with a DOF matrix, because it is impossible to know *a priori* which λs will be visited by the algorithm during its execution.
  2. non-grid evaluation can't have `lambda.selection.lossfunction` = `NULL`,we automatically convert it in `'GCV'`:

     ```
     warning("An optimized method needs a loss function to perform the
         evaluation, selecting 'lambda.selection.lossfunction' as 'GCV'.")
     ```

  3. if we perform non-`NULL` `DOF.evaluation`, it is obvious that we want to use some loss function: we automatically select `lambda.selection.lossfunction='GCV'`:

     ```
     warning("Dof are computed, setting 'lambda.selection.lossfunction' to
         'GCV'.")
     ```

  4. If we pass a `lambda.selection.method='newton'` and not null BC, we will swap for reasons of implementation to the `'newton_fd'` criterion with the warning :

     ```
     warning("'newton' 'lambda.selection.criterion' can't be performed with
         non-NULL boundary conditions, using 'newton_fd' instead.")
     ```

- **Problem specific**. Some parameters are strictly related to some types of optimization or DOF calculations. We raise a warning every time someone:

  1. sets a `DOF.stochastic.seed` or `DOF.stochastic.realizations` different from the default if `DOF.evaluation` is not `'stochastic'`.
  2. defines a different `lambda.optimization.tolerance` from the default whenever the `lambda.selection.criterion` is `'grid'`, since it will not be used.

- **DOF.matrix**. `DOF.matrix` is a powerful tool implemented to avoid the re-computation of Degrees Of Freedom whenever already known, this fastens the execution time exceedingly, as the longest part of any GCV evaluation is always spent in the identification of these quantities. However they are limited in application to pure grid methods and have to be provided for all possible lambdas that are passed for testing. Meaning that there is currently no way to store just a partial number of DOFs specific for some lambdas and not for others. Future projects might think about implementing this feature.

  Passing `DOF.matrix` is the only way to allow `lambda.selection.lossfunction` = "`gcv`" whenever we set (or leave defaulted) `DOF.evaluation` = `NULL`, otherwise an error will be raised:

```
stop("Either DOF_matrix different from NULL or DOF_evaluation different from
    'not_required', otherwise loss_function = 'GCV' can't be computed.")
```

With similar methods we have also modified lower calls to `R` code in files `smoothing_CPP`, `smoothing_manifold_CPP`, `smoothing_manifold_volume_CPP`, the spatio-temporal versions `smoothing_CPP_time`, `smoothing_manifold_CPP_time`, `smoothing_manifold_volume_CPP_time` and their GAM analogous in file named `smoothGAM_CPP`; both reordering inputs and adding checks necessary for the cause, like for example:

```
if(is.null(lambda))
{
   lambda <- vector(length=0)
} else {
   lambda <- as.vector(lambda)}
}
storage.mode(optim) <- "integer"
storage.mode(lambda) <- "double"
DOF.matrix <- as.matrix(DOF.matrix)
storage.mode(DOF.matrix) <- "double"
storage.mode(DOF.stochastic.realizations) <- "integer"
storage.mode(DOF.stochastic.seed) <- "integer"
storage.mode(GCV.inflation.factor) <- "double"
storage.mode(lambda.optimization.tolernce) <- "double"
```

Where `optim` is an internal integer vector that codifies (in `smooth.FEM`) strings contained in `optimization`, `DOF.evaluation` and `lambda.selection.lossfunction` as numbers, to be later decoded at `C++` level, by `OptimizationData` class constructor. How does the translation work?

	0	1	2
lambda.selection.criterion	grid	newton	newton_fd
DOF.evaluation	not_required	stochastic	exact
lambda.selection.lossfunction	unused	GCV	

Table 4.3: `optim` conversion

For instance `optim` = `c(0,1,1)` means grid optimization, with stochastic GCV; `optim` = `c(2,1,0)` means finite differences Newton optimization, with stochastic DOF, but without loss function. That, by the way, is infeasible, since `lambda.selection.lossfunction` is set automatically to `'GCV'` by one of the aforementioned warnings.

4.2 Carrier

The next topic we want to discuss is definitely a key part of our project: variadic template class `Carrier`, contained in `Carrier.h`. We have already mentioned this object in the introduction of

this Chapter, discussing our need to create optimization methods to become a convenient and efficient plugin for any version of the library, we will now introduce how we managed to overcome the difficulties.

4.2.1 Storage efficiency

To be precise, the current idea of the `Carrier` was born after our first porting from version Negri [9] to Colli-Colombo [4]. In that circumstance, we experienced great difficulties in having to rewrite all the syntax to link directly classes of `MixedFERegression` type with our optimization superstructure, having as entry level abstract class `Lambda_optimizer` and its children (developed in detail in Section 4.3). Moreover, we had been informed that [1] might had changed the way of working of many `MixedFERegression` methods, likely besides the fundamental ones. Therefore, we started to envision the possibility to define a *medium* term, assuming the role of a "container", a "carrier", able to store all the data needed for DOF/GCV computation purposes.

Two were the prerequisites for a successful layout: the unique part of the carrier depending from the work of others had to be its construction, while any further optimization structure was to rely just on this term. Form a development standpoint, even though its building was rooted in the collection of terms both from `RegressionData` [RD] and `MixedFERegression` [MFER] class-types, we wanted the storage to avoid copies, possibly without moving terms (which, by the way, is not feasible after the work of [5], see FPIRLS algorithm for further details). As a consequence, we designed the `Carrier` to support a full pointer type syntax, i.e. its content are simply a collection of pointers to `const` and a few function wrappers to the members of `MixedFERegression` that are relevant for our purposes.

This syntax ensures to our class an exceedingly fast construction - since storing a pointer is much cheaper than making a copy of big matrices - without disregarding safety, being the `const` a guarantee to avoid damages on the original data. Finally, a pointer to the original [MFER]-type, defined through template parameter `InputHandler`, was added together with a pointer to `OptimizationData` in order to both make `apply` (Appendix B) methods available and store the data for optimization purposes. Just to complete the picture, we had to modify some interfaces of the aforementioned [RD] and [MFER] classes to support pointer getters, definitely a minor change with respect to what we had to do with the `Lambda_optimizer`.

4.2.2 Variadic inheritance of specializations

The second big issue for an efficient container is the ability to store just what is needed for computations; nothing more, nothing less. This, of course, as a matter of economy, but also for consistency and readability. For this reason we started to think the "carrier" as a base structure from which making possible to derive by inheritance different specializations depending on the type of regression at hand: pointwise data or areal, temporal, presence of forcing terms etc... However, this hope was found quite immediately practically infeasible.

Indeed, as it is easy to see, the single bricks we have described before are not totally separate, inconsistent. Actually it is the contrary: we can and often have to mix them together. We may create an areal problem with forcing terms, or a separable temporal problem with pointwise data, or even more complex structures might be defined in the future as the library will evolve through time, and we wanted to do something that will be able to cope with new extensions. Therefore we decided to change our perspective: instead of making the `Carrier` to be specialized by inheritance, we decided it to variadically inherit the specializations!

The final structure of the `Carrier` is the following:

```
1 template<typename InputHandler, typename... Extensions>
2 class Carrier: public Extensions...
```

where `Extensions...` is a parameter pack, naturally empty for plain regression, but that can be filled with those "bricks" required by the specific problem at hand. As for the moment, two extensions have been implemented: `class Areal`, for areal data, and `class Forced`, for the presence of forcing term, while one is just defined in order to guide the usage of the `Carrier` also for temporal problems: `class Temporal`.

Each one of these "extension classes" contains just the data useful for their specific regression type and can be added to the base layout as variadic `Extensions...` . This produces two major benefits: minimalism and, even more important, the possibility of creating free combinations of the problem types just adding more "bricks" to the plain baseline, whenever needed: e.g. `Carrier` < `InputHandler`, `Areal` > can solve areal problems, while if we define a `Carrier` < `InputHandler`, `Forced`, `Temporal` > we will be able to perform space-time forced smoothing etc...

Just a word about construction, before passing to the real optimization: for the sake of simplicity `Carrier` (and also the possible extensions), can be empty constructed and then filled in a second moment with data flowing from the (in the meanwhile) constructed or updated `RegressionData` or `MixedFERegression`, this work is performed by a global setter named `set_all`, called during the process of `Carrier` building in `Regression_Skeleton.h`. In particular these builders and their products are defined in Table 4.4.

	`Carrier`	`Areal`	`Forced`
`build_plain_carrier`	X		
`build_areal_carrier`	X	X	
`build_forced_carrier`	X		X
`build_forced_areal_carrier`	X	X	X

Table 4.4: `Carrier Extension...` building convention

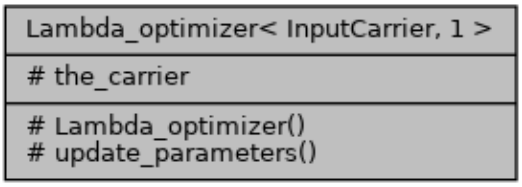
4.3 Lambda Optimization

Being responsible for naming the entire Chapter, template `Lambda_Opimizer` and its derived children classes are without doubt some of the most important structures present in our project, and a litmus test of our concepts of portability, minimalism and efficiency. This class hierarchy, contained in header file `Lambda_Optimizer.h`, suitably fed by a `Carrier`, is meant to compute Degrees Of Freedom, generalized cross validation and possibly even other quantities of interest, like predictions in the locations, errors and variance of the phenomenon at hand. Let's start from discussing the code structure:

```
1 template <typename InputCarrier, UInt size>
2 class Lambda_optimizer
```

`Lambda_optimizer` depends on two template parameters: `InputCarrier` which is the specific type of - possibly extended - `Carrier` that will store just the essential data for optimization, according to the problem at hand, and `size`, which is an integer representing the dimension of the λ to be used. We have not really described this concept in detail - we defer to [1], for further insight on these topics - but the optimization parameter is uni-dimensional for spatial regression problems or parabolic spatio-temporal smoothing. However, it becomes bi-dimensional in the so called separable spatio-temporal regression problems. Being our project pioneering, we only had to conceive optimization for the first of these cases. Nonetheless, for the sake of extensibilty and in coherence with our way of working, we decided to conceive this class from its origin as a flexible structure, able to be generalized even to multidimensional contexts: this is the reason why we kept the `size` parameter in the template, and its hierarchy.

But, since our project has only coded uni-dimensional spatial regression optimization, we will only work in the partially specialized setting where `size` = 1, and study in detail the case. Moreover, from now on, we will also make use of class diagrams, that will help us in understanding synthetically the content of the objects we are going to analyze. Starting from `Lambda_optimizer`:



we can see it being almost empty, since the content will come gradually in the derived children (and grandchildren).
The only `protected` term stored inside is the `InputCarrier` `the_carrier`, that will be used as support by the whole family, a constructor which takes only an `InputCarrier` as parameter and a `virtual` function named `update_parmeters` that will update all the parameters needed for computation. The presence of a `virtual` function makes the entire structure a *template virtual class*. We will soon see how this member will be specialized by the derived classes. [1]

4.3.1 `GCV_Family`

The only template class that, as for the moment, inherits from `Lambda_Opimizer` is `GCV_Family`. This term is the (still `vitual`) base class for all optimization methods relying on the computation of the Generalized Cross Validation, see Section 3.1 for insights. Parallel to this structure, in the future, other developers might create implementations to perform K-fold cross validation or other type of statistical validation techniques, see Section 7.3 for further detail.

```
1 template <typename InputCarrier>
2 class GCV_Family<InputCarrier, 1>: Lambda_optimizer<InputCarrier, 1>
```

At this level the main data related to output are computed and stored: predicted value in the locations, estimate of the residuals, sum of square of the aforementioned residuals, estimated variance of the error and, of course, the Degrees Of Freedom. However, not all the terms can be really made available without further additions. For instance predictions and Degrees Of Freedom might be computed with more efficient techniques according to different strategies, e.g. in an *exact* way (i.e. computing the value directly) or via *stochastic* estimation, see Section 3.1.3 if you are not familiar with these kind of topics. For that reason, this intermediate class is still just a virtual tool, that can be declined in several ways according to the specific DOF computation strategy chosen. As for our implementation, also taking a moderate dose of inspiration from the work of the past, we have decided to adapt the two strategies cited before, giving birth to:

```
1 template<typename InputCarrier>
2 class GCV_Exact<InputCarrier, 1>: public GCV_Family<InputCarrier, 1>
```

for the *exact* computation, while leaving the stochastic one in:

```
1 template<typename InputCarrier>
2 class GCV_Stochastic<InputCarrier, 1>: public GCV_Family<InputCarrier, 1>
```

grandchildren of the original `Lambda_Optimizer`, true objects used by the optimization methods that we will describe in the next Sections.

Concerning the main methods implemented at level `Regression_Family` not much has to be

[1] Note that all classes in this Section are contained in header file `Lambda_Optimizer.h`, if not differently specified.

said, they are mainly short utility tools related to the evaluation of the fundamental quantities described before and a few brick updaters that will be used by their derived classes in their main updaters. The only function of moderate importance is:

```
void compute_z_hat_from_f_hat(const VectorXr & f_hat)
{
  if (this->the_carrier.has_W())
  {
     this->z_hat = (*this->the_carrier.get_Hp())*(*this->the_carrier.get_zp()) +
         this->the_carrier.lmbQ((*this->the_carrier.get_psip())*f_hat);
  }
  else
  {
     this->z_hat = (*this->the_carrier.get_psip())*f_hat;
  }
}
```

that allows to derive the value of the predicted value in the locations starting from solving the main system discussed in Sections 1.3 and 1.4, where `f_hat` is exactly the homonym first block of the system solution.

Note: This code snipped gives us the chance to discuss an important method present in classes of `MixedFERegression`-type, that we have passed as a wrapper caller into the `Carrier`: `leftMultiplyByQ`, in `Carrier` synthesized (without too much fantasy) by the acronym `lmbq`. This function uses the typical decomposition of projection matrices seen in Section 1.3 to speed up the computation of the products by Q. We have directly tested the time employed by the code with and without this feature and proved, first hand, the efficiency given by this tool. Therefore tip for novice developers: if you have to multiply something by matrix Q, *always* do that through this fancy utility and you won't regret the benefit in execution time.

4.3.2 `GOF_updater` (Goodness Of Fit)

Getting back to the class hierarchy, the two children of `GCV_Family` are pretty much dissimilar, both in the quantities computed and - whenever evaluating the same terms - also in the order in which values of interest are updated. In particular, the structure of *exact* GCV computation allows not only to evaluate exactly (without prejudice concerning numerical errors in matrix inversion) the value of the function depending on λ, but also to derive the term one or more times in an efficient way, this will be the key for the implementation of Newton's method. Conversely, stochastic computation of GCV is generally much faster (as we will see in Chapter 5), but slightly less accurate and there are no tools developed so far to estimate precisely the derivatives of the GCV function.

The key point to understand what we will mention from now on is efficiency. Since the calculations done at this level for DOF estimation, prediction of values in the locations, evaluation of GCV and further derivatives etc... are very time consuming operations and represent (besides main system factorization and solution of Section 1.3) the bottleneck of temporal efficiency, updates of these quantities of interest - i.e. the modifications regarding just the ones changing from one λ and another - have to be done only when strictly necessary, possibly in the best order, thus saving time and memory. For this reasons, being GCV evaluation and its derivatives our main concern, we have defined three functions called `updaters`, that have the purpose of performing, in the most efficient way possible at our disposal, the identification of all and only the quantities strictly necessary for the definition of the aforementioned derivatives. Adopting the convention that each updater is named after the derivative it tries to maintain: in order to compute the first GCV derivative we will first have to call the `first_updater`. However, being updaters hierarchical, the `zero_updater` (namely everything needed to identify the GCV, given a certain λ) will have to be called before. Similarly the `second_updater` will be called just when

the first two will have been performed, and only if a second derivative will be required.

Hence, it's easy to see that the updating strategy is very sequential but also pretty much delicate: each updater is written in the most essential way possible and has to be called only if we need to compute specific quantities: e.g. if we are using a finite differences Newton method on an *exact* DOF evaluation, we will never need to compute first or second derivatives. So the terms just needed for their evaluation, produced in the corresponding updaters, will never be called. On the other hand, if we have already computed the value of GCV for a certain λ and thus called the `zero_updater`, it would be a waste of precious time to recompute all the data before, e.g., applying the `second_updater`, since the GCV second derivative required is for the *same* value of λ. In order to address this somehow chaotic, but we rather prefer calling it minimal, efficiency driven puzzle, instead of flooding our classes with over-complicated call architectures; we decided to delegate the whole burden of the updating to an external class named `GOF_updater` - where GOF stays for Goodness Of Fit.

The purpose of this tool, contained in header `GOF_Updater.h`, is storing the right updaters from the `GCV_Family` derived classes and then managing an efficient and elegant performance of the required refreshing concerning quantities of interest, at the right time.

```
template <typename LambdaOptim, typename T>
class GOF_updater
```

The structure of this class is really simple, given the type of `lambda` (scalar or vector) in `T` and the `LambdaOptim` optimization method on which to work: its content consists of just three members:

```
std::vector<T> last_lambda_derivatives;
std::vector<std::function<void(Real)>> updaters;
LambdaOptim * start_ptr = nullptr;
```

The first term is a vector storing the last value of lambda for which the corresponding updater has been called (e.g `last_lambda_derivatives` = [2., 1., -1.]) means that the last lambda for which `zero_updater` was called is 2, `first_updater` 1 and `second_updater` was never used, as -1 is the default initialization for never being called. `updaters` is simply the collection of the original updaters from `GCV_Exact` or `GCV_Stochastic`, copied in a vector of functions and stored at `GOF_updater` level. Finally, for reasons of safety concerning possible copies or address modifications `start_ptr` stores the address of the corresponding object on which the update has to be performed. Finally, the main function of this class is represented by:

```
inline void call_to(UInt finish, T lambda, LambdaOptim * lopt_ptr)
{
  if(start_ptr != lopt_ptr) // new pointer to be stored (or first time we store)
  {
     initialize(std::vector<Real>{-1.,-1.,-1.}); // dummy initialize the last lambdas
     updaters_setter(lopt_ptr); // set all the updaters from the given pointer
     start_ptr = lopt_ptr; // keep track of pointer to avoid this procedure next time
  }

  bool found = false; // cycle breaker
  for(UInt i = 0; i<=finish && found==false; ++i) // loop until the desired level
     if(lambda != last_lambda_derivatives[i])
     {  // found the first not updated derivative
        // update from that derivative to the needed level (finish)
        call_from_to(i, finish, lambda);
        found = true; // break the cycle since the update is complete
     }
}
```

This method first ensures to be working on the right pointer and in case of change it re-initializes the `updaters` and `last_lambda_derivatives`. Then, given as parameter `finish` the actual derivative order to be updated, based on the hierarchy, checks if all the lower orders have already been called. Once it finds the first non updated term, it stars calling the updaters in sequence, from that element onward, until the desired derivative order: `call_from_to(i, finish, lambda)`.

4.3.3 `GCV_Stochastic` and `GCV_Exact`

In this Subsection we don't really want to give all the implementation details of the construction of classes `GCV_Stochastic` and `GCV_Exact`, it would be almost impossible. Moreover all information required to understand these methods can already be found commented directly in the code, to allow novice as well as expert users to understand step-by-step why and how things are being done. Some of the code, moreover, has also been partially adapted from previous versions, e.g. [9] or [4]. What we would like to discuss are our contributions and how we have managed to make the computations as qualitative as possible through `Carrier`-driven SFINAE.
Disclaimer. The following modifications are very technical, please continue to read in two cases: interest in the reason behind the constructs used or better understanding the efficiency of some choices. It is highly suggest to read the following content with, at least, in parallel the class diagram of appendix C and Sections 3.1.1-3.1.2 or even the code itself. It would be advisable to have scanned the code at least once to better grasp the meaning of some of the following modifications.

- Single evaluation of the stochastic matrix `US` in *stochastic* computation and `R` in the *exact* counterpart, performed directly in the constructor for immediate storage and avoiding useless repetitions (and very time consuming).
- In version [5] passing the `DOF_matrix`, see Section 4.1.1, was possible to avoid DOF computation, we have reactivated this possibility allowing `GCV_Stochastic` to evaluate the GCV through these values. Why *stochastic* and not *exact*? Because it had an updating system, with less requirements concerning matrix multiplication or inversion, so just for reasons of efficiency.
- Most of the matrices necessary to compute derivatives, prediction in locations etc..., can be derived via formulas that are *highly problem specific*, see e.g formula 3.1, where we might need an extra term for the computation if the problem presents forcing term, otherwise not. Here, the variadic nature of the `Carrier` becomes really useful. Indeed, we have created a struct made only of `static` member functions called `AuxiliaryOptimizer` - present in file `Auxiliary_Optimizer.h` - whose role is exactly that of storing functions that provide different compile time implementations depending on the type of carrier used (stored in `Lambda_Optimizer` and thus passed down to derived classes).

```
1 template<typename InputCarrier>
2 static typename
      std::enable_if<std::is_same<multi_bool_type<std::is_base_of<Forced,
      InputCarrier>::value>,t_type>::value, UInt>::type
3   universal_z_hat_setter(VectorXr & z_hat, InputCarrier & carrier, const
        MatrixXr & S, AuxiliaryData<InputCarrier> & adt, const Real lambda);
4
5 template<typename InputCarrier>
6 static typename
      std::enable_if<std::is_same<multi_bool_type<std::is_base_of<Forced,
      InputCarrier>::value>,f_type>::value, UInt>::type
7   universal_z_hat_setter(VectorXr & z_hat, InputCarrier & carrier, const
        MatrixXr & S, AuxiliaryData<InputCarrier> & adt, const Real lambda);
```

As we can see from the function declarations, these terms use SFINAE as basic tools to select the correct function in each case. In this particular example the compiler will check

if the `Carrier` passed to the `Lambda_Optimizer` will inherit from `Forced`, if affirmative it will instantiate the first function, otherwise the second, and in any case no error will ever be issued. The integer return type might even be used to handle exceptions, but we have not felt the need to implement them, at least for how direct the structure is now.

Note: `multi_bool_type` is a Boolean parameter pack created to address the problem of generating a compile time array of Boolean of unspecified length, necessary to deal with the *a priori* indefinite number of possible extensions that can be added to a `Carrier`, some typedef are also provided and the whole template structure is contained in the general header of the library `FdaPDE.h`.

```
template <bool ... b>
struct multi_bool_type
{};
```

```
typedef multi_bool_type<true> t_type;
typedef multi_bool_type<false> f_type;
typedef multi_bool_type<true, true> tt_type;
typedef multi_bool_type<false, true> ft_type;
...
```

- As specified in Section 2.5, we have corrected a very severe error concerning the factorization of a non-symmetric matrix mistaken for one of the kind, under areal data. In a standard setting we would be forced to chose to use always the factorizer for non-symmetric matrices, not being able to detect *a priori* which problem will have to be solved, until run time. However, the `Carrier` extension type can be used to effectively bypas this criticality at compile time, using Cholesky factorization for pointwise data and LU for the areal case simply by means of a type trait. Here is the new factorizer declaration that can be found in `AuxiliaryOptimizer` :: `universal_V_setter`, where `V` is a matrix used for *exact* GCV evaluation.

```
typedef typename std::conditional<std::is_base_of<Areal, InputCarrier>::value,
    Eigen::PartialPivLU<MatrixXr>, Eigen::LDLT<MatrixXr>>::type Factorizer;
```

- A SFINAE based support struct called `AuxiliaryData` has also been implemented to cleverly keep track of all the useful quantities computed in the different phases of the updates, in order to apply in the most responsive way possible the set of computational rules described in Sections 3.1.1-3.1.2, with high-performance storage of the byproducts recyclable in further evaluations. This factor will be a key ingredient to not worsen grid evaluation times w.r.t. the old versions of the library in testing phase, see Chapter 5 for details.

- `apply` function in `MixedFERegression` has been adapted as seen in appendix B, to be compliant with the `Lambda_Optimizer`, as well as a second `apply_to_b` has been added to ensure avoiding useless re-building re-factorization of the already defined system matrix in case of *stochastic* DOF evaluation. We have also independently corrected some issues, fixed in [1] concerning erroneous application of boundary conditions.

- Thanks to the work described in Chapter 3, we have been able to fully make symmetric the expression for `Forced` GCV and the standard one, thus enabling them to be computed with the same formula once the (differently computed) bricks a, b, c are ready. For reasons of generality - still not having studied what might happen in `Temporal` case - we have kept a fake SFINAE (`true` = `true`) instantiation for the evaluation of GCV and derivatives. If in the future the same formulas will be extended to all cases, it will become feasible to remove these placeholder terms.

4.3.4 The `output_Data` struct

In this Section we describe the structure of the struct `output_Data`, which is implemented in the file `src/Lambda_Optimization/Include/Solution_builders.h`. It contains all the information necessary to be returned to `R`:

```
struct output_Data
{
        std::string           content{"Empty"};
        MatrixXr              z_hat;
        std::vector<Real>     rmse;
        Real                  sigma_hat_sq   = -1.0;
        std::vector<Real>     dof            = {};
        Real                  lambda_sol     = 0.0;
        UInt                  lambda_pos     = 0;
        UInt                  n_it           = 0;
        Real                  time_partial   = 0.0;
        std::vector<Real>     GCV_evals      = {-1};
        std::vector<Real>     lambda_vec     = {-1};
        Real                  GCV_opt        = -1;
        int                   termination    = -2;
        MatrixXv              betas;
};
```

Differently from the previous version of the code, we need to have more variables: in particular, we return the computed **z**, which were not passed back in the old versions, the number of iterations - in case of optimization methods - and the time necessary to perform the only optimization method, after the instantiation of the necessary matrices and variables: thus why it is called `time_partial`. Moreover, `termination` is an integer code indicating the reason for the end of the iterations for the optimization procedure: 1 for the reached given tolerance, 2 for maximum iterations, -1 for an error status and -2 if we do not use any optimization method. Then we have the other members, already present in the previous code, such as $\hat{\sigma}^2$, the values of $\hat{\beta}$, the best values of GCV and corresponding λ, the vector of λ's and the vector of all the GCV. We notice that these vectors can represent two different things: if we are in the case of simple vector evaluation (see Section 4.5), they represent exactly the vector of the evaluated GCV and the vector of the corresponding λ. If instead we are in the case of an optimization method, these vectors will be filled with the sequence of GCV and λ visited during the iterative procedure (see Sections 4.6.1 and 4.6.2).

4.4 The Function Wrapper class

We now pass to describe the classes which are used to perform function evaluations and optimization procedures: they were not present in the previous versions of the code and they are built in order to be reused also in more general contexts than the GCV function evaluation. Namely, we start from the most important class, the class which implements a general function, which gives the possibility of evaluating a function at different points and computing first and second derivatives.

This class, called in the code `Function_Wrapper` and contained in the header file named as `src/Lambda_Optimization/Include/Function_variadic.h`, is a variadic template that might take a variable number of arguments. In particular, it has the following fixed arguments:
`<typename Dtype, typename Ctype, typename Tuple, typename Hessian>`
representing, respectively, domain type, image type and image type of the gradient (or Jacobian) and the Hessian of the function implemented. Indeed, we leave the possibility of using function from $\mathbb{R}^n$ to $\mathbb{R}^m$, with n and m generic integers.

Then we have another argument: `typename... Extensions`, which means, if properly used, that the class can inherit from another one or not. This is very useful, because, as in our case of the GCV computation, if another class is able to compute the function and the derivatives, then this wrapper class can make possible to generalize other cases of study. Indeed, it can also be built from a function given in the constructor, instead of inheriting from another class. In particular, to reach this objective we have two constructors, in which we can specify the possible inheritance:

```
//! Constructor taking object Extensions and initializing with it the new class
template<typename D=typename std::enable_if<sizeof...(Extensions)!=0,void>,
   typename ...T>
Function_Wrapper(T&& ...ext):Extensions(std::forward<T>(ext))...{};

//! Since I have defined a constructor I need to indicate the default constructor
Function_Wrapper() = default;
```

After having built the class, we left two possibilities. We start from the first, which is that this class does not inherit from any other class. In this case, we can set the following three members contained inside the class:

```
std::function<Ctype(Dtype)> g;
std::function<Tuple(Dtype)> dg; //derivative
std::function<Hessian(Dtype)> ddg; //second_derivative
```

which represent the function and its derivatives, by means of the setter:

```
//! Function to inizialize the g,dg,ddg, used only if you explicit directly the
    functions
inline void set_functions(const std::function<Ctype(Dtype)> & g_,const
    std::function<Tuple(Dtype)> & dg_,const std::function<Hessian(Dtype)> & ddg_)
{
       g   = g_;
       dg  = dg_;
       ddg = ddg_;
}
```

In this way the class contains all the information to evaluate a function and its first and second derivatives. If we have set this functions, thus not inheriting from a class which computes the values externally, we can exploit SFINAE, namely we define the following three methods:

```
template <typename U>
   typename std::enable_if<sizeof...(Extensions)==0 ||
            std::is_void<U>::value, Ctype>::type
   evaluate_f(U lambda)
   {
          return g(lambda);
   }

//!First derivative
template <typename U>
   typename std::enable_if<sizeof...(Extensions)==0 ||
            std::is_void<U>::value, Tuple>::type
   evaluate_first_derivative(U lambda)
   {
          return dg(lambda);
   }
// is_void<U> is always false for us,
// used to make the deduction argument and SFINAE work
```

```
//! Second derivative
template <typename U>
   typename std::enable_if<sizeof...(Extensions)==0 ||
           std::is_void<U>::value, Hessian>::type
   evaluate_second_derivative(U lambda)
   {
          return ddg(lambda);
   }
```

By means of the function `std::enable_if`, we instantiate these functions only if no `Extensions` are provided, i.e., the class does not inherits from anything. Notice the dummy presence of a template argument U, which has the only role of permitting the SFINAE mechanism. Indeed, in our cases the check `std::is_void<U>::value` will be always false, because the function is intended to be only numeric, thus `lambda` given in input is always numeric and not of type `void`: the only part which is important is the one related to the `Extensions`. This first type of use is not the one we exploited in our project, but it was proposed as a generalization if, for example, there were the necessity of having a function in a class by giving the analytical expression of it, together with the first and second derivatives.

The second type of use, which is the one we exploit in the GCV computations, is to make this class inherit from another one, which already evaluates the function and its derivatives, by means of the three members: `compute_f,compute_fp,compute_fs`. Indeed, in this case again by SFINAE we instantiate the following three functions:

```
template <typename U>
   typename std::enable_if<sizeof...(Extensions)!=0 ||
           std::is_void<U>::value, Ctype>::type
   evaluate_f(U lambda )
   {
          return this->compute_f(lambda);
   }

//! First derivative
template <typename U>
   typename std::enable_if<sizeof...(Extensions)!=0 ||
           std::is_void<U>::value, Tuple>::type
   evaluate_first_derivative(U lambda)
   {
          return this->compute_fp(lambda);
   }

//! Second derivative
template <typename U>
   typename std::enable_if<sizeof...(Extensions)!=0 ||
           std::is_void<U>::value, Hessian>::type
   evaluate_second_derivative(U lambda)
   {
          return this->compute_fs(lambda);
   }
```

In this case clearly `Extensions` is not empty and the check returns true. We can see that the function has the only role of calling the inherited method. Now, either the class `Function_Wrapper` inherited from another one or not, it contains three members with the same names
`evaluate_f, evaluate_first_derivative, evaluate_second_derivative`
which can be called to get the evaluation of a function and its derivatives.

This construction seems uselessly complicated, but it is very useful for another reason: since this class is intended to be used by a class performing some kind of optimization, and since in general

it could inherit from any class which evaluates some kind of functions, we can easily include this class as a member of the optimization class without troubling about the template parameters of the father classes of the `Function_Variadic`. To stick to our example, we need to build two classes which compute GCV: the first one with the stochastic GCV and the second one with the exact GCV. If we had not used the class `Function_Wrapper`, we should have built at least two versions for each optimization class, one for each kind of GCV computation. By means of the wrapper we can build only one otpimization class independently of the inherited classes of the Wrapper: all this information in encapsulated in `Extensions`.
We can now pass to describe the classes which make use of the Function Wrapper.

4.5 The grid evaluation method

In the previous versions of the code there was only one method to find the best value of λ for the GCV computations: given a vector of λ, it computed the GCV at every point of the grid, and then compute the minimum between the values, corresponding to the best λ. This is a naive way of proceeding, depending on the choice of the vector of λ to evaluate.
In order to preserve this functionality, leaving the user the possibility of choosing between this and the optimization methods, we created the class `Vec_evaluation`. Since it is intended to have a member of type `Function_Wrapper`, it is a template class with the same arguments as the wrapper:

```
template <typename Tuple, typename Hessian, typename... Extensions>
```

The class contains as members the wrapper, which is called for the function evaluations, and the vector of `Tuple` λ's which are intended to be used in the evaluation. The constructor of the class reads as follows:

```
Vec_evaluation(Function_Wrapper<Tuple, Real, Tuple, Hessian, Extensions...> & F_,
    const std::vector<Tuple> & lambda_vec_): F(F_), lambda_vec(lambda_vec_) {}
```

As we can see, we simply initialize the two members of the class. Then we have three methods: the first two are present as virtual functions: the two functions do nothing, but we decided to avoid the use of pure virtual functions, for the sake of generality, since there could be some cases in which they are not needed and thus this base class could be instantiated as it is. These methods are the following:

```
virtual void compute_specific_parameters(void) {};
virtual void compute_specific_parameters_best(void) {};
```

They will be overridden in the specific class for GCV computation.
In conclusion, there is another method which is intended to be general and reusable by any other class which needs to compute an evaluation of this kind.

```
std::pair<std::vector<Real>, UInt> compute_vector(void)
```

The return type is the pair of the vector of function evaluations corresponding to the vector of λ's and the unsigned integer value corresponding to the index in the vector of the λ with minimum GCV value. This class is intended to work for functions from $\mathbb{R}^n \to \mathbb{R}$, thus we only talk about scalar values as output, whereas we leave the template argument `Tuple` for the input values. The method is straightforward:

```
UInt dim = lambda_vec.size();
UInt index_min = 0; //Assume the first one is the minimum
std::vector<Real> evaluations(dim);

for (UInt i=0; i<dim; i++)
{
```

```
        this->F.set_index(i);
        evaluations[i] = this->F.evaluate_f(this->lambda_vec[i]);
        this->compute_specific_parameters();
        if (i==0)
             this->compute_specific_parameters_best();
        if (evaluations[i]<evaluations[index_min])
        {
            this->compute_specific_parameters_best();
                index_min=i;
        }
}

return {evaluations,index_min};
```

We notice that two different possibilities of computing particular parameters are left to be characterized (if present): function `compute_specific_parameters` allows to compute values which are needed at each evaluation, whereas `compute_specific_parameters_best` permits to update the quantities only to the value with the current minimum value (indeed it is called only if the current function value is less than the current optimum found). Having found the minimum, the aforementioned pair with the function evaluations and the index of the best λ are returned. Now, we build another class called `Eval_GCV`, which inherits from the previous one, but it computes the evaluations only in GCV case, which is a function from $\mathbb{R}$ to $\mathbb{R}$. The interesting part of this class is that it overrides two methods, because we need some extra computations at each evaluation, namely we need the value of the Degrees Of Freedom (DOF) and the rmse. Thus the method `compute_specific_parameters` becomes:

```
void compute_specific_parameters(void) override
{
        this->F.set_output_partial();
}
```

It calls a function which is present in the wrapper, because this class is built having in mind that the Wrapper `F` should inherit from one of the GCV classes (`GCV_family`) previously described, which have this method. Indeed, the method `set_output_partial` is defined in the file `src/Lambda_Optimization/Include/Lambda_Optimizer_imp.h` and it only adds the current DOFs and [rmse] to a vector at each evaluation. These vectors will be used to build the output to communicate to `R`. Then, at each evaluation, whenever a better GCV value than the current minimum is found, we need to perform some other computations, namely we need to save the values of the $\hat{\tilde{\mathbf{z}}}$ and $\hat{\sigma}^2$. Then we exploit the function `compute_specific_parameters_best`:

```
void compute_specific_parameters_best(void) override
{
        this->F.set_output_partial_best();
}
```

This method calls a method present in `F`, because, as before, `F` is thought to be inheriting from `GCV_family`. This method, defined again in `Lambda_Optimizer_imp.h`, has the role of saving the correct values in the `output` members of `F`. Then, the method `compute_vector` is exactly as described in the father class. In conclusion, we have also another method in the class: the method `Get_optimization_vectorial`. In this method, we call the function `compute_vector` and then we copy the `output_Data` from the `F`, which is now completed with all the needed information, and we complete it by adding to the respective members the returned values from the method `compute_vector`: we take into account the vector of GCV evaluations, the vector of λ evaluated, the best GCV value, the best λ and its position in the vector given, already with the `R` notations, i.e. starting from 1 the numeration of the elements of a vector:

```
output_Data Get_optimization_vectorial(void) //! Output constructor
{
	std::pair<std::vector<Real>, UInt> p = this->compute_vector();
	output_Data output=this->F.get_output_full();
	output.GCV_evals = p.first;
	output.lambda_sol = this->lambda_vec.at(p.second);
	output.lambda_pos = 1+p.second; //in R numbering
	output.lambda_vec = this->lambda_vec;
	output.GCV_opt   = p.first.at(p.second);

	return output;
}
```

The method then returns the `output` completed with the desired evaluations, and it will be called in the file `Regression_Skeleton.h` in order to perform the computations and then build the output to be returned to `R`. The `R` parameter to be set if we want a grid (i.e., vectorial) evaluation is `optimization="grid"`, together with a parameter called `lambda`, which has to be filled with the vector of λ's to be evaluated.

4.6 Optimization Class

We now describe the new classes we added to perform iterative optimization procedures, alternative to the simple vector evaluation already present in the code and described in the previous Section. The class for the Newton's methods is declared in the file `Newton.h` and defined in `Newton_imp.h`. It is a class to implement the method described in the introductory Sections, with the computation of the exact derivatives of the function. The same goes for the Newton's Finite differences method. Since they have a lot of structures in common, the two methods are built inheriting from a common virtual father class, called `Opt_methods`, which contains the Function Wrapper as a member, and the pure virtual method `compute` which performs the iterative procedure, and will be overridden.

```
template <typename Tuple, typename Hessian, typename... Extensions>
class Opt_methods
{
	protected:
		//! Contructor
		Opt_methods(Function_Wrapper<Tuple, Real, Tuple, Hessian,
		    Extensions...> & F_): F(F_) {}
	public:
		//! Function to apply the optimization method and obtain as a result
		    the couple (optimal lambda, optimal value of the function)
		Function_Wrapper<Tuple, Real, Tuple, Hessian, Extensions...> & F;
		virtual std::pair<Tuple, UInt> compute (const Tuple & x0, const Real
		    tolerance, const UInt max_iter, Checker & ch, std::vector<Real> &
		    GCV_v, std::vector<Real> & lambda_v) = 0;
};
```

Again it is a template class, because we leave the possibility of using scalars or vector valued functions. We can appreciate the utility of the Function Wrapper: with an easy declaration we can have a member computing whatever function we need. The constructor takes as input only a reference to the wrapper: the optimization class will modify this member during the optimization procedure. In header `src/Lambda_Optimization/Include/Newton.h` we build some other auxiliary classes which are used in the function `compute`. In particular, we created a class `Checker`, which has the only role of checking if the termination of the optimization method was due to the reaching of maximum iterations or to the reaching of the prescribed tolerance.

Moreover, we build another struct, called `Auxiliary`: in general it is built for a generic template argument `Tuple`, but we fully specialized it in the two cases we are interested in: scalar numbers (`double`) or vectors, using the `Eigen::Matrix<Real,Eigen::Dynamic,1>`, renamed by `typedef` as `VectorXr`. The structs are the following

```
template <typename Tuple>
struct Auxiliary
{
	//NOT yet implemented
};

//! Auxiliary class to perform elementary mathematical operations and checks: specialization for 1 dimensional case
template<>
struct Auxiliary<Real>
{
	public:
		Auxiliary(void) {};

		//! Check if the input value is zero
		static inline bool isNull(Real n) {return (n == 0);}
		//! Apply a division
		static inline void divide(Real a, Real b, Real & x){x = b/a;}
		//! Compute the norm of the residual
		static inline Real residual(Real a) {return std::abs(a);}
};

//! Auxiliary class to perform elementary mathematical operations and checks: specialization for n dimensional case
template<>
struct Auxiliary<VectorXr>
{
	public:
		Auxiliary(void) {};

		//! Check if the input value is zero
		static inline bool isNull(MatrixXr n)
		{
			UInt sz = n.size();
			return (n == MatrixXr::Zero(sz,sz));
		}
		//! Solve a linear system in the optimization method
		static inline void divide(const MatrixXr & A, const VectorXr & b, VectorXr & x)
		{
			Cholesky::solve(A, b, x);
		}
		 //! Compute the norm of the residual
		static inline Real residual(VectorXr a)
		{
			return a.norm();
		}
};
```

Actually, in our code only the scalar version is used, but in view of other extensions, for example, in the time-varying case, the argument of the function to be optimized are more than one. This structs have the role of computing divisions (or solving linear systems in the vectorial cases), computing a modulus (norms, respectively) and checking if the parameter is zero, to avoid division by zero. We can now pass to discuss the two Newton's classes, which do nothing but override the aforementioned virtual method `compute`.

4.6.1 Exact Newton's method

Starting from the Exact Newton's method, form the declaration

```
template <typename Tuple, typename Hessian, typename ...Extensions>
class Newton_ex: public Opt_methods<Tuple, Hessian, Extensions...>
```

we see that it inherits from the previously described class `Opt_methods`. The template arguments are the same as described in the previous Section. The constructor takes as input the reference to the Function Wrapper, which is passed to the constructor of the father class. No other members are used in this implementation. We describe the main function of the class, the method `compute`, which overrides the corresponding method of the father class. The declaration is as following:

```
std::pair<Tuple, UInt> compute (const Tuple & x0, const Real tolerance, const UInt
    max_iter, Checker & ch, std::vector<Real> & GCV_v, std::vector<Real> & lambda_v)
    override;
```

The first parameter is the initial guess λ_0, the second is the tolerance, which can be decided by the user, the third one is the maximum number of iterations, which in our case will be set to 40 by default. Then there is a reference to an object of class `Checker` previously mentioned: it is used to set the termination reason of the iterative procedure. In conclusion, there are two references to `std::vector`: these vectors will be filled by the values of the parameter λ and the corresponding GCV visited in the iterative procedure. The return type is a `std::pair` which contains the number of iterations and the optimal value of λ computed. Analyzing the definition of the method, in `Newton_imp.h`, we can find the procedure described in the theoretical Sections.

```
// Initialize the algorithm
Tuple x_old;
Tuple x     = x0;
UInt  n_iter = 0;
Real  error  = std::numeric_limits<Real>::infinity();

Real valmin, valcur, lambda_min;
UInt Nm = 6;
std::vector<Real> vals={5.000000e-05, 1.442700e-03, 4.162766e-02, 1.201124e+00,
    3.465724e+01, 1.000000e+03};
valcur = this->F.evaluate_f(vals[0]);
lambda_min = 5e-5;
valmin = valcur;
for(UInt i=1; i<Nm; i++)
{
      valcur = this->F.evaluate_f(vals[i]);
      if(valcur<valmin)
      {
              valmin = valcur;
              lambda_min = vals[i];
      }
}
if(x>lambda_min/4||x<=0)
{
      x = lambda_min/8;
}
```

In the first phase we initialize the method, considering six possible initial values, choosing the one with the best GCV, `lambda_min`, and then dividing it by 8. We also give to the user the possibility to select an initial value: in order to be reasonably sure that this choice won't spoil the convergence, we make the user-given initial value accepted only if it is positive and smaller than one fourth of `lambda_min`. Then, we have the second "computing" phase of the method: after printing the initial value of λ, for the user's interest, we perform the effective iteration

procedure, printing at each repetition the current iteration and the value of the residual.

```
Rprintf("\n Starting Newton's iterations: starting point lambda=%f\n",x);
Real   fx  = this->F.evaluate_f(x);
Tuple  fpx = this->F.evaluate_first_derivative (x);
Hessian fsx = this->F.evaluate_second_derivative(x);

while(n_iter < max_iter)
{
      GCV_v.push_back(fx);
      lambda_v.push_back(x);
      if(Auxiliary<Tuple>::isNull(fsx))
      {
             return {x, n_iter};
      }
      ++n_iter;
      x_old = x;
      Auxiliary<Tuple>::divide(fsx, fpx, x);
      x = x_old - x;
      if (x<=0)
      {
             Rprintf("\nProbably monotone increasing GCV function\n");
             fx = this->F.evaluate_f(x);
             return {x_old, n_iter};
      }
      fpx = this->F.evaluate_first_derivative (x);
      error = Auxiliary<Tuple>::residual(fpx);

      Rprintf("\nStep number %d of EXACT-NEWTON: residual = %f\n", n_iter, error);

      if(error<tolerance)
      {
             ch.set_tolerance();
             fx = this->F.evaluate_f(x);
             return {x, n_iter};
      }
      fx  = this->F.evaluate_f(x);
      fsx = this->F.evaluate_second_derivative(x);
 }
fx = this->F.evaluate_f(x);
ch.set_max_iter();
return {x, n_iter};
```

This code snippet is exactly as described in the theoretical part, apart from a detail: since in general the method is supposed to converge to some minimum, due to the shape of GCV function, we consider the case of reaching a non-positive value of λ as a problem due to the shape of the function. In particular we print (see , e.g., [19] for a reference on printing in `R`) the message: `Probably monotone increasing GCV function`, because in this case we expect the minimum to be as near as possible to zero (see test in Section 5.1.4 for an application of this case). We notice that this function also works in the case of functions from $\mathbb{R}^n$ to $\mathbb{R}$ due to the specialization of the class `Auxiliary`. In conclusion, the method checks the stopping criterion and if it is satisfied, the pair made of the number of iterations and the optimal λ are returned, after having called the `Checker` method `set_tolerance`, which sets the reason of the end of the iterations as cause by the reached tolerance (value 1). If the tolerance is never reached, the same pair is returned, but in this case the `Checker` sets, by means of the method `set_max_iter`, the cause of stopping the iterations in order to mean that the maximum number of iterations was reached (value 2).

4.6.2 Finite differences Newton's method

The second optimization method we implemented is the Newton's method with the approximation of the derivatives by means of finite differences, as described in Section 3.2. Again this new class inherits from the father class `Opt_methods`, as we can see in its declaration:

```
template <typename Tuple, typename Hessian, typename ...Extensions>
class Newton_fd: public Opt_methods<Tuple, Hessian, Extensions...>
```

The constructor takes as input the reference to the Function Wrapper, which is passed to the constructor of the father class. We notice that in the case of Finite differences the implemented method can be applied only to functions from $\mathbb{R}$ to $\mathbb{R}$. Indeed, in the case of vectorial functions, we need to approximate the Jacobian matrix and not only the one-directional derivative. We then specialize the general class (we recall that we are using `C++11`), not implemented, in the one we are interested in, endowing it with the overridden method:

```
std::pair<Real, UInt>
   compute (const Real & x0, const Real tolerance, const UInt max_iter, Checker & ch,
       std::vector<Real> & GCV_v, std::vector<Real> & lambda_v) override;
```

In a future version of the code, the new class for the vectorial case could be easily built inheriting again from the `Opt_methods` class: this is the reason for such a general version of the father class. The above mentioned method `compute` overrides the one of the father class: the iterative procedure is exactly the same as in the exact case, thus we only notice that the computation of the derivatives is approximated by mean of centered finite differences (see Section 3.2 for details). In particular, in the code we fix a value for `h=4e-6` and we compute:

```
Real fxph = this->F.evaluate_f(x+h);
Real fxmh = this->F.evaluate_f(x-h);
Real fx = this->F.evaluate_f(x);
Real fpx = (fxph-fxmh)/(2*h);
Real fsx = (fxph+fxmh-(2*fx))/(h*h);
```

By using only three evaluations of the function, in our case the GCV function, we are able to compute the approximated first derivative, `fpx`, by means of centered finite differences, and the second derivative, `fsx`, again by means of centered finite differences. We print the current iteration and the current residual, in order to make it clear to the user; clearly, now the printed message reads:

```
Rprintf("\nStep number %d of FD-NEWTON: residual = %f\n", n_iter, error);
```

The remaining part of the method is very close to the Exact Newton's method, thus we refer to Section (4.6.1) for a more detailed description.

4.6.3 Optimization Methods Factory

In order to choose and instantiate the correct type of optimization method, we built the following factory, which can be found in the file `Optimization_Method_Factory.h`:

```
template<typename Function, typename Tuple, typename Hessian, typename
    EvaluationType>
class Opt_method_factory
{
  public:
       static std::unique_ptr<Opt_methods<Tuple,Hessian,EvaluationType>>
           create_Opt_method(const std::string & validation, Function & F)
       {
               if(validation=="newton")
```

```
                return make_unique<Newton_ex<Tuple, Hessian,
                    EvaluationType>>(F);
        if(validation=="newton_fd")
                return make_unique<Newton_fd<EvaluationType>>(F);
        else // default is finite differences
        {
           Rprintf("Method not found, using Newton_fd");
           return make_unique<Newton_fd<Tuple, Hessian, EvaluationType>>(F);
        }
    }
};
```

If in the `R` code the user inserts, at the input `optimization=`, the keyword `newton`, the factory builds a unique pointer to the above mentioned class `Newton_ex`, implementing the exact Newton, whereas if the users inserts the word `newton_fd`, the factory returns a pointer to the aforementioned class `Newton_fd`. In all the other cases the strategy implements finite differences. In this context we notice the importance of the `overridden` method `compute`. The return type of the factory will be of the common father type, as we can see from the declaration:

```
static std::unique_ptr<Opt_methods<Tuple,Hessian,EvaluationType>>
    create_Opt_method(const std::string & validation, Function & F)
```

but, due to the keyword `virtual`, the method `compute` is called according to the type of the derived class.
We recall that, since the library is compiled by means of `C++11`, the function `std::make_unique` is not implemented (only from `C++14`) and thus, in the file `src/Global_Utilities/Include/Make_Unique.h` (see the Appendix for the code organization in folders), we implement the following function, which has the same action:

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
      return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

In conclusion, it is important to notice that the factory can be easily extended by simply adding other optimization methods: the presence of the Function Wrapper simplifies a lot the instantiations and it is enough to inherit from `Opt_methods` class to build another method to be returned by the factory as a pointer to the method itself.

4.7 Output

Having discussed how evaluations are performed and the optimum is found according to the different strategies, we want to conclude describing results storage and transfer back to `R`. As in the previous versions, all calls to the different methods are performed in function `regression_skeleton`, contained in header `Regression_Skeleton.h`, now having a few subroutines to allow `Carrier` instantiation, definition and computation of the optimization methods and output production. We will not cover entirely this description since it would consist simply in code translation of all the actions we have already defined through the previous Sections, for anyone interested in the syntax the suggestion is that of looking directly to the file, which is already heavily commented to guide the interested reader through the single steps.

Instead, we will show in greater detail the output management. Having increased substantially the number of parameters passed back to the user, we felt the need to somehow disrupt the symmetry with other skeletons and extract the output production in a separate routine. As said before, purpose of this new function, stored in namespace `Solution_Builders`, will be that of

preparing the output to be read by R. A strong piece of advice for future developers-readers, if new optimization methods will be implemented for other types of regression or FPCA, we highly suggest to follow our structure and implementation, in order to restore the parallelism. Getting back to the definition:

```
namespace Solution_Builders // Unique class to manage the output
{
  template<typename InputHandler, UInt ORDER, UInt mydim, UInt ndim>
  static SEXP build_solution_plain_regression(const MatrixXr & solution, const
      output_Data & output, const MeshHandler<ORDER, mydim, ndim> & mesh, const
      InputHandler & regressionData);
};
```

given the solution matrix produced by `MixedFERegression apply` methods, the same aforementioned `output_Data` struct described in Section 4.3.4, mesh and `RegressionData`, this static - to avoid the storage of an irrelevant empty object - method produces a `SEXP`, namely an `R` readable generic type that will finally be processed in the final lines of function `smooth.FEM` to produce the `R` list that the user will examine at the end of the call. In this final Section we will see in detail how this `SEXP` is built and processed by `R`.

Besides some minor pre-processing on regression coefficients, if present, what follows are the first lines of the output builder. We will discuss them just to point out the syntax used and later we will add a Table, to describe the content of each single parameter returned to `R`. As can be seen in the code snippet, in order to prepare the solution to be built, it's necessary to initialize an empty `SEXP` as `NILSEXP`, the default code for null.

Just a little note, *never* return a `NILSEXP` to `R` since it will be counted as a failure in the execution of the function, producing the dreadful and hardly interpretable shutdown of `Rstudio` working session, with the message `R session aborted`. Returning to the code, the `result` is then defined as a vector containing 22 slots. Note that within this notation a vector is not required to contain objects of the same type. To better grasp the idea, if you are not familiar with `R`, `C++` analogous of `VECSXP` is probably `std :: tuple`. Then, we start filling the first four slots of the `VECSXP` inserting, respectively, two matrices, a vector and a real.

```
//Copy result in R memory
SEXP result = NILSXP;
result = PROTECT(Rf_allocVector(VECSXP, 22));

SET_VECTOR_ELT(result, 0, Rf_allocMatrix(REALSXP, solution.rows(), solution.cols()));
Real *rans = REAL(VECTOR_ELT(result, 0));
for(UInt j = 0; j < solution.cols(); j++)
{
  for(UInt i=0; i<solution.rows(); i++)
     rans[i + solution.rows()*j] = solution(i,j);
}

SET_VECTOR_ELT(result, 1, Rf_allocMatrix(REALSXP, output.z_hat.rows(),
    output.z_hat.cols()));
rans = REAL(VECTOR_ELT(result, 1));
for(UInt j=0; j<output.z_hat.cols(); j++)
{
  for(UInt i = 0; i < output.z_hat.rows(); i++)
     rans[i + output.z_hat.rows()*j] = output.z_hat(i,j);
}

UInt size_rmse = output.rmse.size();
SET_VECTOR_ELT(result, 2, Rf_allocVector(REALSXP, size_rmse));
rans = REAL(VECTOR_ELT(result, 2));
```

```
for(UInt j=0; j<size_rmse; j++)
{
  rans[j] = output.rmse[j];
}

SET_VECTOR_ELT(result, 3, Rf_allocVector(REALSXP, 1));
rans= REAL(VECTOR_ELT(result, 3));
rans[0] = output.sigma_hat_sq;
```

Note: C++ and R differ in array indexing, i.e. while C++ vectors start with first component being id. 0, the first element of an R vector is number 1. For this reason, when building the VECSXP, even though the object has to be read by R, the first cell of the tuple will be defined as 0. However, once returned the result to R, leaving the object unmodified, the first component is going to be addressed as element 1! Remember this shift in notation, otherwise some information might be lost or misinterpreted in the conversion. Moreover, these errors are very hard to detect, and there have been projects highly focused in their identification, e.g. Colli-Colombo [4], where you might find more insight about conversion between C++ and R syntax.

Closed this digression, we would like to show explicitly how matrices are stored to be passed into R. In the first line of the second block we prepare the space required (rows and cols) for the matrix: SET_VECTOR_ELT(result, 0, Rf_allocMatrix(REALSXP, solution.rows(), solution.cols())); After that, we define a pointer that coincides with the first element of the given structure and start to fill the components with a standard double cycle on rows and columns. Note that, since working in pointer-like syntax, we need to know that the matrix storage is performed "by rows".

Otherwise, trying to write the transposed of the true matrix, we would end up in another obscure R session aborted, in case of it being rectangular. In the unfortunate hypothesis of it being squared no error would be issued, and so finding the problem might be even more tricky. Finally in the third and fourth block we see how to fill a vector - exactly like a matrix, but just one cycle is required for the storage - or a single number. In the three cases the common denominator is always the pointer like syntax and the need to pre-allocate the space and type needed for the storage. An error in these terms won't likely be detected at compile time, but will surely cause R session aborted during the first run. We close the discussion with full Table 4.5 of the content returned to R.

Last, we would like to show how the output, passed to R is managed in function smooth.FEM. As we can see in the code, first we collect data coming from regression depending on the problem type we are facing, then we create a list called solution mixing all the data that can be used to derive the coefficients of our model.

```
# Save coefficients for regression
if(!is.null(covariates))
{
  if(lambda.selection.method == 'grid' & is.null(DOF.evaluation) &
      is.null(loss_function))
  {
     beta =  matrix(data=bigsol[[15]],nrow=ncol(covariates),ncol=length(lambda))
  } else {
     beta = matrix(data=bigsol[[15]],nrow=ncol(covariates),ncol=1)
  }
} else {beta = NULL}
```

```
# Build solution
solution = list(
  f = bigsol[[1]][1:numnodes,],
  g = bigsol[[1]][(numnodes+1):(2*numnodes),],
  z_hat = bigsol[[2]],
  beta = beta,
  rmse = bigsol[[3]],
  estimated_sd = bigsol[[4]])
```

ID[R]	FORMAT	CONTENT
1	matrix	If no optimization has to be performed, column k is the estimate of $[f, g]^T$ in the nodes according to the k-th λ produced by the user, if GCV has to be computed it is just a vector containing the solution in correspondence of the best λ in the grid or produced by the optimization method
2	matrix	Same as 1, but returns the estimate of the function in the locations: $\hat{z}$
3	vector	Same as 1 substituting coefficient to column, returns an estimate of the mean square error in the locations
4	real	Only meaningful under DOF computation, returns an estimate of the variance of noise ε: $\hat{\sigma}^2$
5	real	Only meaningful under GCV computation, returns the value of λ minimizing GCV
6	real	Only meaningful under GCV computation, returns the value of the best GCV found by the optimization algorithm or among the λ in the grid
7	integer	Only meaningful under GCV computation, returns the position of the best λ in the vector passed by the user
8	integer	Only meaningful under optimized methods, returns the number of iterations to ensure convergence
9	integer	Code: returns the type of termination
10	integer	Code: returns the type of optimization performed
11	vector	Degrees Of Freedom (if computed) of all the λ passed by the user in grid or visited by the optimization algorithm in optimized methods
12	vector	Vector of λ passed by the user in grid or visited by the optimization algorithm in optimized methods
13	vector	Vector of GCV for each λ passed by the user in grid or visited by the optimization algorithm in optimized methods
14	real	Time spent in solving the problem, not counting `preapply` (Appendix B)
15	matrix	Same as 1, but returns the estimate of regression coefficients in each column
16-20	//	Multiple objects related to tree search handling, see [8] for further detail, we have kept both order and notation
21-22	//	Multiple objects related to `bary.coordinates`, see [8] for further detail, we have kept both order and notation

Table 4.5: Output content

Then we create a second list collecting all the data related to optimization and we also register execution time of the optimized procedure, converting in strings the codes of termination and optimization.

```
# Collect optimization data
optimization = list(
  lambda_solution = bigsol[[5]],
  lambda_position = bigsol[[6]],
  GCV = bigsol[[7]],
  optimization_details = list(
     iterations = bigsol[[8]],
     termination = bigsol[[9]],
     optimization_type = bigsol[[10]]),
  dof = bigsol[[11]],
  lambda_vector = bigsol[[12]],
  GCV_vector = bigsol[[13]])

# Register time
time = bigsol[[14]]
```

Skipping the part where mesh and barycenters are produced, since part of [8], we finally prepare the return list to be passed back to the user, under name of `reslist`.

```
# Make Functional objects object
fit.FEM = FEM(solution$f, FEMbasis)
PDEmisfit.FEM = FEM(solution$g, FEMbasis)

# Prepare for returning
reslist = list(
  fit.FEM = fit.FEM,
  PDEmisfit.FEM = PDEmisfit.FEM,
  solution = solution,
  optimization = optimization,
  time = time,
  bary.locations = bary.locations)

return(reslist)
```

Chapter 5

Simulation Studies

In this Chapter we present some simulation studies, to make a comparison between the evaluation of GCV by means of the old vectorial grid and our implemented optimization methods. The tests used are in the files `tests/smooth.FEM.2D.tests.R`, `tests/smooth.FEM.2.5D.tests.R` and `tests/smooth.FEM.3D.tests.R`.
We consider a mesh with N nodes. The n observations can coincide to the mesh nodes or not. Data are generated from the already presented model $z_i = \mathbf{w}_i^T \boldsymbol{\beta} + f(\mathbf{p}_i) + \varepsilon_i,\ i = 1, \dots, n$, or from (1.10) for the areal smoothing, where f is a suitable test function selected according to the geometry of the mesh, $\boldsymbol{\beta}$ is fixed so that $W\boldsymbol{\beta}$ has a range compatible with the range of the test function, W being the already described matrix of the covariates. The error term ε_i is chosen as a Gaussian noise with 0 mean and variance 0.05 $range(W\boldsymbol{\beta} + f(\mathbf{p}_i))$, i.e. 5% of the range of the field, including the contribution of the covariates (if any). In order to compare the performances of the different methods, we will use the following definition of RMSE:

$$RMSE^2 = \frac{\sum_{i=0}^{N}(f_i - \hat{f}_i)^2}{N},$$

where f_i is the evaluation of f at the mesh nodes, where $\hat{f}_i$ are computed, and we expect this value to be as small as possible. The GCV method seems reasonably, if not theoretically, to be a good way of finding the smoothing parameter λ leading to a small value of RMSE. All the tests are performed on a PC with Ubuntu 18.04, with 16GB of RAM, with 25 repetitions for each test.

For what concerns the methods, in the following we write

- `grid_ex(`n_λ`)`=Grid evaluation of exact GCV given the following vector of n_λ values of λ:

```
lambda = 10^seq(-7,2,length.out=n_lambda)
```

  where n_λ is chosen to be in general 20, whereas we increase it a bit in the cases when the optimization methods take much more time of computation, in order to compare the RMSE of a grid evaluation taking the same time with those other methods.

- `newt_ex` = Exact Newton's method with computation of exact GCV (the initial guess is left to automatic initialization). If not specified, the tolerance is set to 0.05, whereas, in case of the presence also of tolerance 0.005, the strings become: `newt_ex_0.05` or `newt_ex_0.005`.

- `newt_fd_ex`= Finite differences Newton's method, with computation of exact GCV (the initial guess is left to automatic initialization).

- `grid_st(`n_λ`)`=Grid evaluation of stochastic GCV given the following vector of n_λ values of λ:

```
lambda = 10^seq(-7,2,length.out=n_lambda)
```

- `newt_fd_st` = Finite differences Newton's method with stochastic GCV (the initial guess is left to automatic initialization). If not specified, the tolerance is set to 0.05, whereas, in case of the presence also of tolerance 0.005, the strings become: `newt_fd_st_0.05` and `newt_fd_st_0.005`.

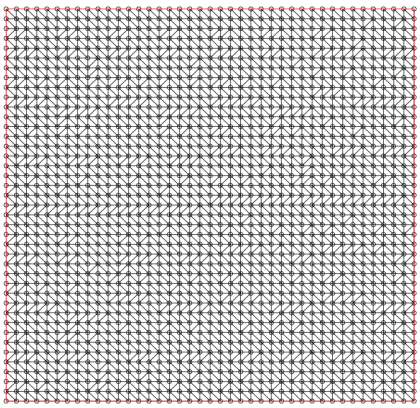
5.1 2D smoothing

5.1.1 Square domain

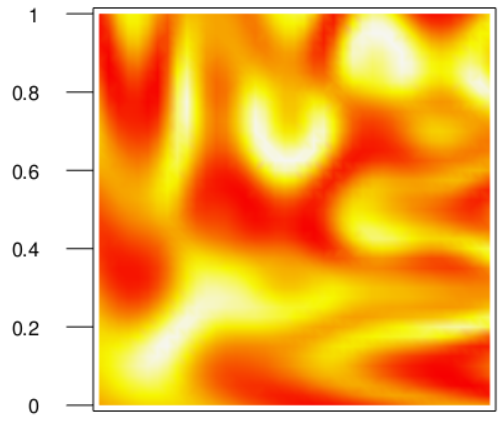
The exact function used in this test is:

$$\begin{aligned} coe(u,v) =& \frac{1}{2}\sin(5\pi x)\exp(-x^2)+1 \\ f(x,y) =& \sin(2\pi(coe(y,1)x\cos(z-2)-y\sin(z-2)))\cdot \\ & \cdot\cos(2\pi(coe(y,1)x\cos(z-2+\frac{\pi}{2})+coe(x,1)y\sin((z-2)\frac{\pi}{2}))) \end{aligned}$$

Covariates are not present.
We increase the number of nodes at each case: locations where observations are sampled coincide with the nodes.



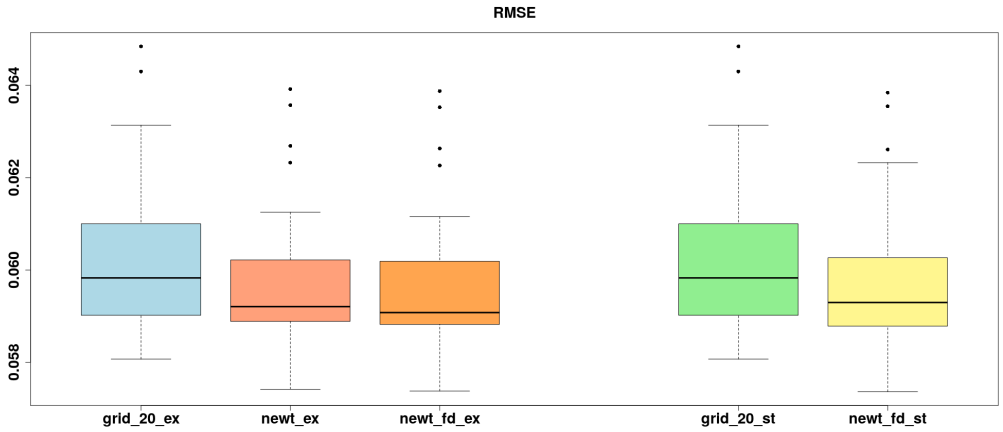
(a) Base square mesh refined across tests.



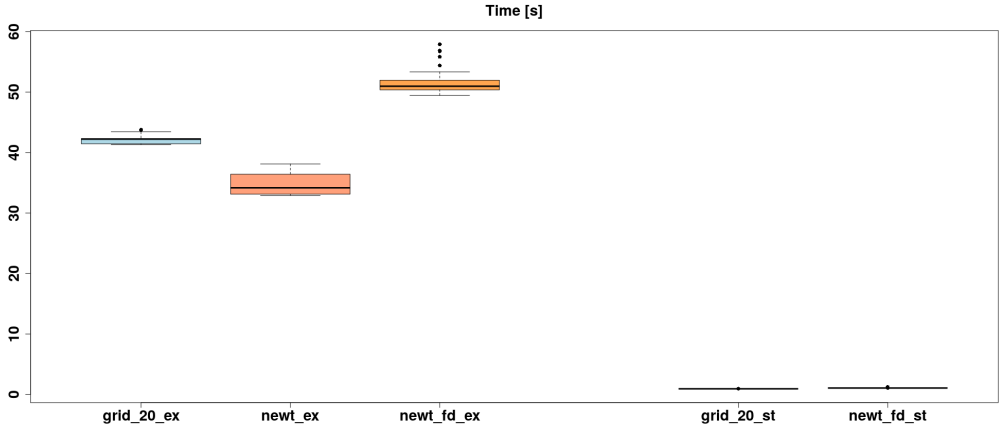
(b) Representation of the not perturbed function f.

Figure 5.1: Square domain: mesh nodes 1'681 - observations 1'681, locations are nodes

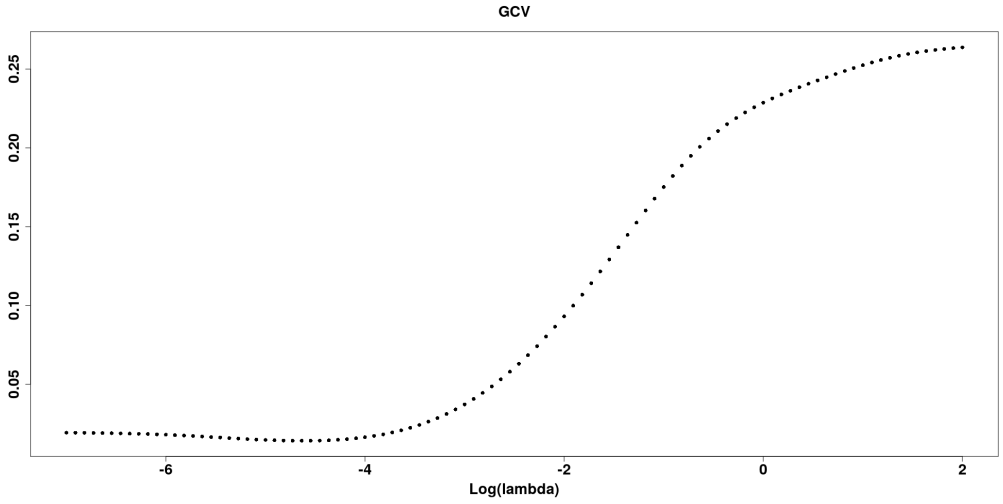
Square domain - Test 1 [1'681 nodes - 1'681 observations - point locations at nodes]
Analyzing Figure 5.2, we can observe that Newton's methods have better performances than grid evaluations with 20 values of λ, at least in terms of RMSE. Comparing the computational times, we notice that the grid method with GCV stochastic takes approximately the same time as Newton's finite differences still with stochastic GCV, but with worse RMSE. The same goes for grid and Newton's methods with exact GCV: the latter consumes less time and obtains a smaller RMSE value. We point out that, since the computation of the exact GCV derivatives is highly optimized, the exact Newton's method performs much better than its finite differences counterpart, in terms of time consumption. In conclusion, we notice that the optimal λ value is very similar in all the methods, as we could expect: as we can see in Figure 5.3, we notice that all the represented fields are very similar to each others.



(a) Method vs RMSE



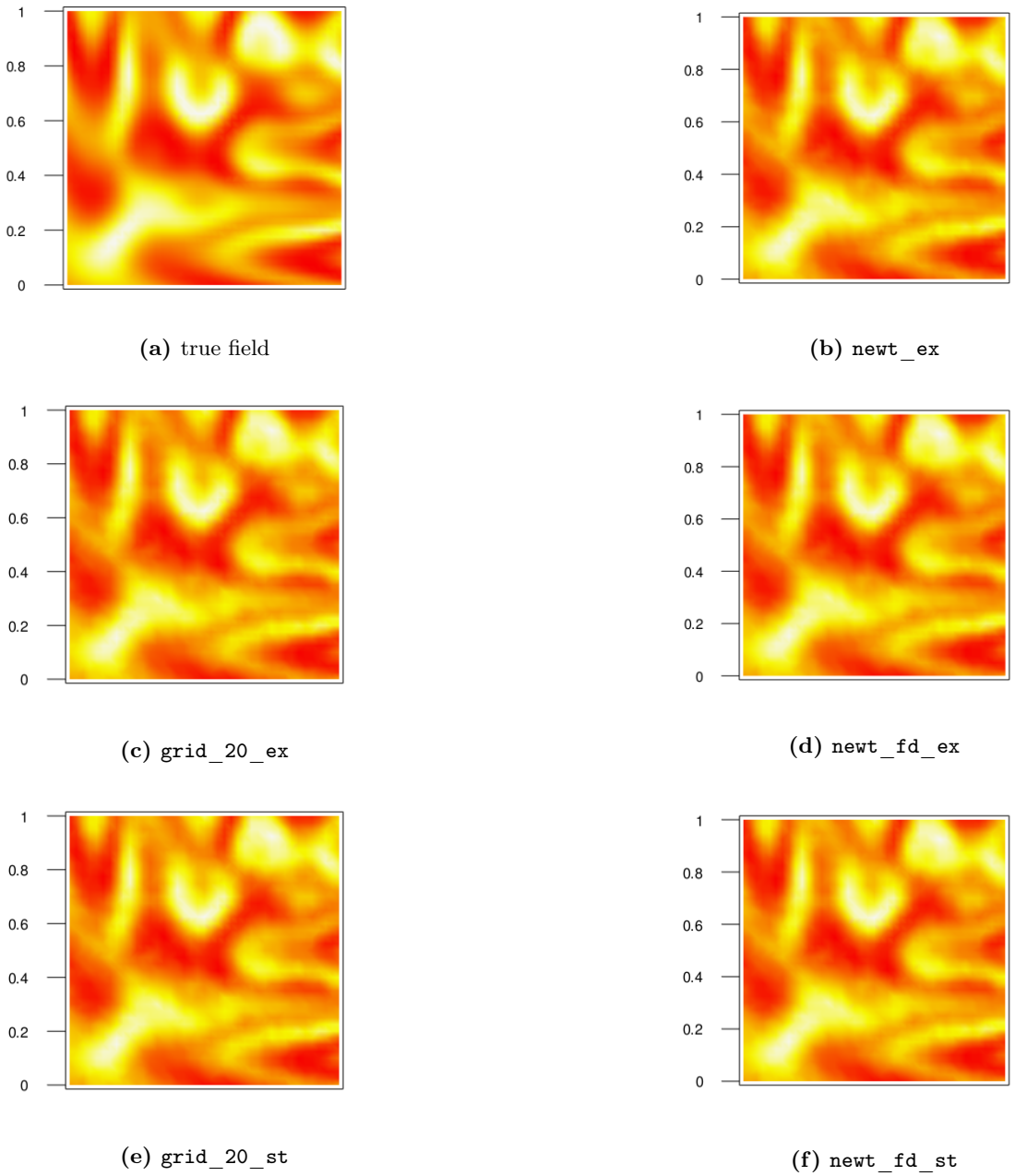
(b) Method vs Time



(c) GCV as function of λ

Figure 5.2: Test 1 (square domain), 1'681 nodes - 1'681 observations- point locations at nodes

Method	Optimal λ	GCV	Average time
grid_20_ex	2.34e-05	1.484e-02	42.22
newt_ex	2.51e-05	1.484e-02	34.17
newt_fd_ex	2.51e-05	1.484e-02	50.96
grid_20_st	2.34e-05	1.484e-02	0.953
newt_fd_st	2.55e-05	1.481e-02	1.085



(a) true field

(b) `newt_ex`

(c) `grid_20_ex`

(d) `newt_fd_ex`

(e) `grid_20_st`

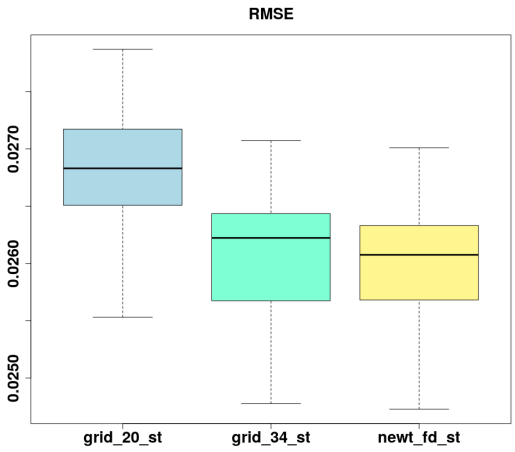
(f) `newt_fd_st`

Figure 5.3: Test 1 (square domain): comparison among the different output fields

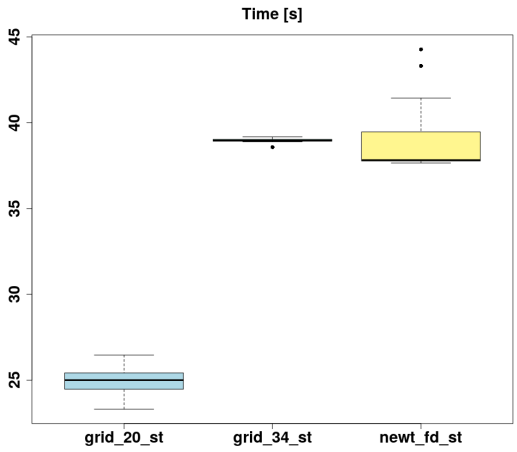
Square domain - Test 2 [3'281 nodes - 3'281 observations - point locations at nodes]
If we test a mesh with a higher number of nodes, i.e., 3'281, we notice that we obtain approximately the same results of Test 1 (for a reference, see Appendix D).

Square domain - Test 3 [15'741 nodes - 15'741 observations - point locations at nodes]
In this case, as we can see in Figure 5.4, we did not compute the exact GCV, since it should be computationally too demanding for any user of the library. We concentrate on stochastic GCV computation. We can observe that Newton's methods have again better performances than grid evaluation strategies with 20 values of λ, in terms of RMSE. Comparing the computational times, we observe that the grid with GCV stochastic takes less time (14 seconds less) than Newton's finite differences with stochastic GCV, but with worse RMSE. If we use a number $n_\lambda = 34$, which is enough to make the grid evaluation method take approximately the same computational time as the Newton's one, we observe again that the RMSE is better in the Newton's case.

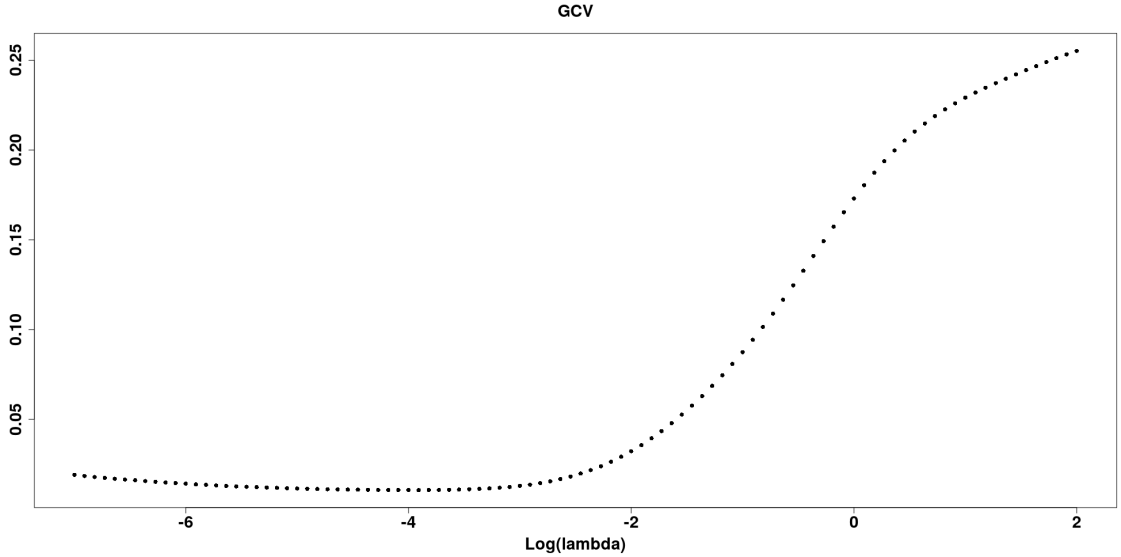
We can conclude that the Newton's finite differences strategy is a better criterion for the choice of the optimal λ.



(a) Method vs RMSE



(b) Method vs Time



(c) GCV as function of λ

Figure 5.4: Test 3 (square domain), 15'741 nodes - 15'741 observations - point locations at nodes

Method	Optimal λ	GCV	Average time
grid_20_st	6.95e-05	1.073e-02	24.41
grid_34_st	1.00e-04	1.484e-02	38.97
newt_fd_st	1.10e-04	1.069e-02	37.20

Square domain - Test 4 [30'971 nodes - 30'971 observations - point locations at nodes]
If we test a mesh with a higher number of nodes, i.e., 30'971, we notice that we obtain approximately the same results of Test 3 (for a reference, see Appendix D). We only remark that using a tolerance of 0.005 or 0.05 for the Newton's method does not change the result: we are led to choose 0.05 as the default value, if not changed by the user.

5.1.2 Mesh C (horseshoe)

In this Section we analyze the following test: namely, let $a(x, y)$ and $d(x, y)$ be the functions:

$$a(x,y) = \begin{cases} \frac{\pi}{4} + x & \text{if } x \geq 0, y > 0 \\ -\frac{\pi}{4} - x & \text{if } x \geq 0, y \leq 0 \\ -\frac{1}{2}\arctan(\frac{y}{x}) & \text{if } x < 0 \end{cases}$$

$$d(x,y) = \begin{cases} -\frac{1}{2} + y & \text{if } x \geq 0, y > 0 \\ -\frac{1}{2} - y & \text{if } x \geq 0, y \leq 0 \\ \sqrt{x^2 + y^2} - \frac{1}{2} & \text{if } x < 0 \end{cases}$$

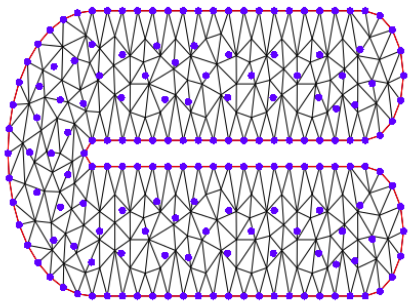
and then

$$f(x, y) = a(x, y) + d(x, y)^2,$$

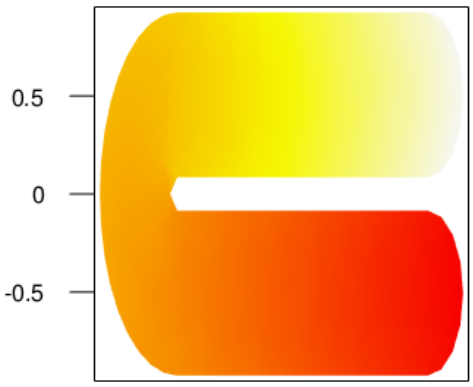
which is a finite area test function based on one proposed by Simon Wood (2008, [17]). The covariates are defined as:

$$\begin{cases} \mathbf{w}_{1i} \sim \mathcal{N}(\mu = 1, \sigma = 2) & i = 1, \ldots, n \\ \mathbf{w}_{2i} = \sin(x_i) & i = 1, \ldots, n \\ \boldsymbol{\beta} = [2, -1]^T, \end{cases}$$

which are the same as in Section 5.1.2. We now represent the mesh and field f in Figure 5.9.



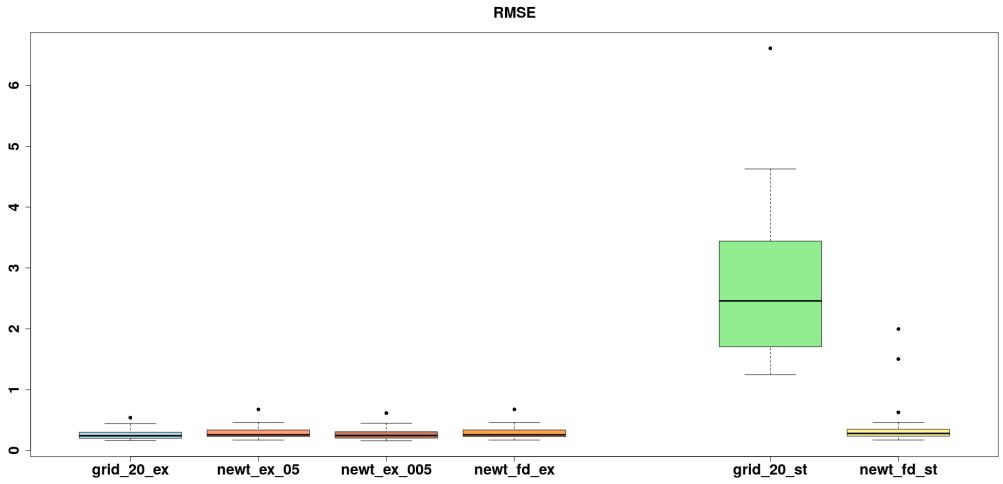
(a) Computational mesh and sample locations.



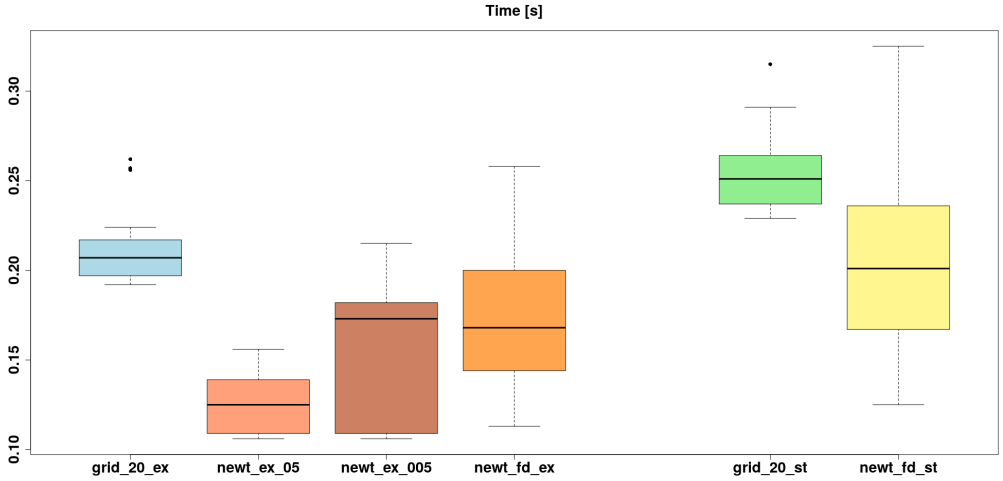
(b) Representation of the not perturbed function f.

Figure 5.5: Mesh C: mesh nodes 264 - observations 171

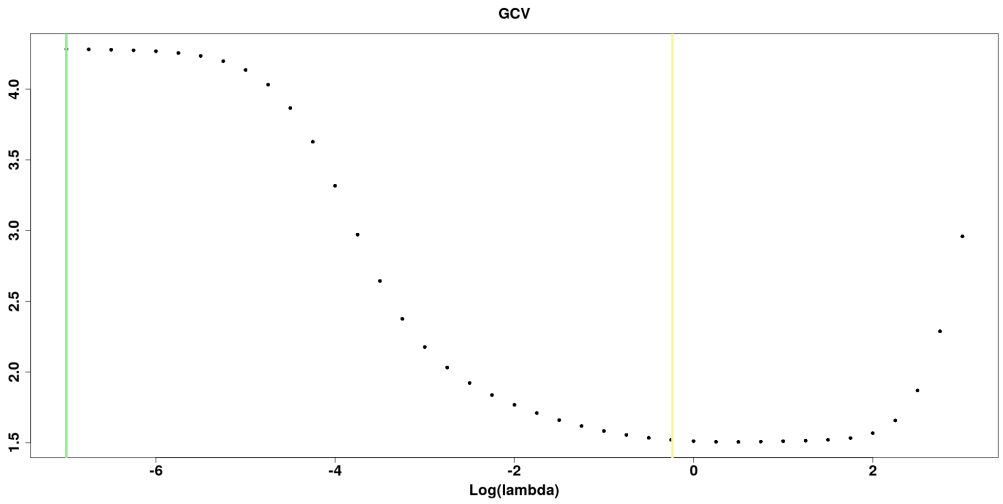
Mesh C - Test 5 [264 nodes - 171 observations - 2 covariates - point locations not at nodes] We can observe in Figure 5.6 that, in this case, the GCV function has an almost flat neighborhood around the minimum, therefore we do not expect excellent perfomances of Newton's methods. Indeed, we see that they are almost equivalent. Interesting to notice that the grid stochastic strategy is not reliable, since for very small values of λ the stochastic GCV is not correct (see Figure 5.7d for the resulting wrong estimation): conversely the stochastic Newton's finite differences seems to be more stable to this kind of inconvenient, as we can see from the boxplot. Also in Figure 5.6c we observe that the average λ chosen in the stochastic grid evaluation method is very different from the one chosen by the Newton's method, which is also the nearest to the minimum of the GCV function.



(a) Method vs RMSE



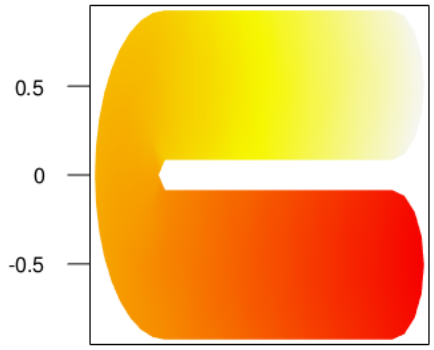
(b) Method vs Time



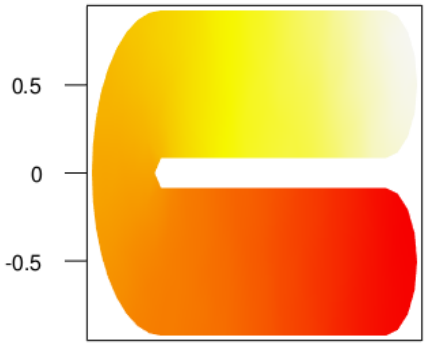
(c) GCV as function of λ: the colored lines represent the optimal λ in the stochastic grid and Newton's method.

Figure 5.6: Test 5 (mesh C), 264 nodes - 171 observations - 2 covariates - point locations not at nodes

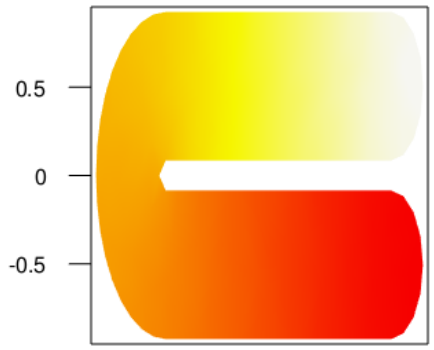
Method	Optimal λ	GCV	Average time
grid_20_ex	3.80e+00	1.55e+00	0.21
newt_ex_0.05	0.60e+00	1.56e+00	0.13
newt_ex_0.005	1.35e+00	1.55e+00	0.17
newt_fd_ex	0.60e+00	1.56e+00	0.17
grid_20_st	1.00e-07	2.64e-04	0.30
newt_fd_st	0.58e+00	1.49e+00	0.20



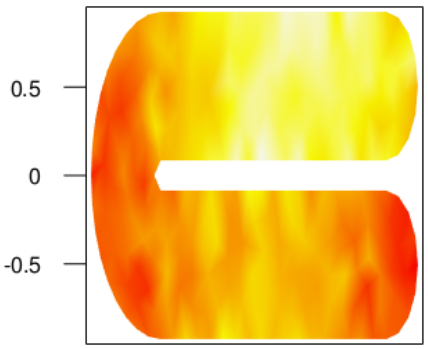
(a) true field



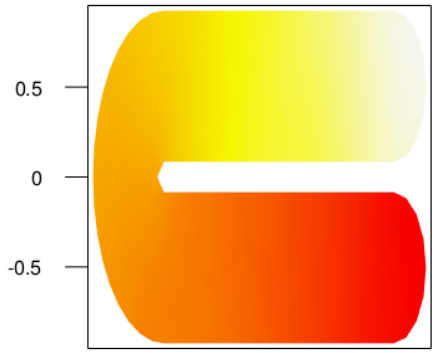
(b) `newt_fd_st`



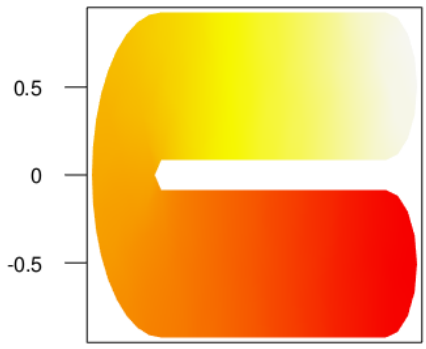
(c) `grid_20_ex`



(d) `grid_20_st`: incorrect



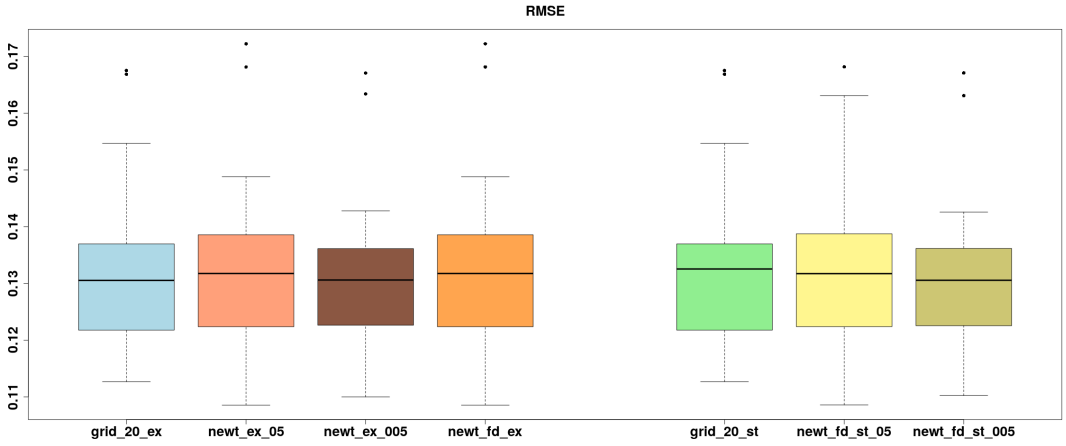
(e) `newt_ex_0.05`



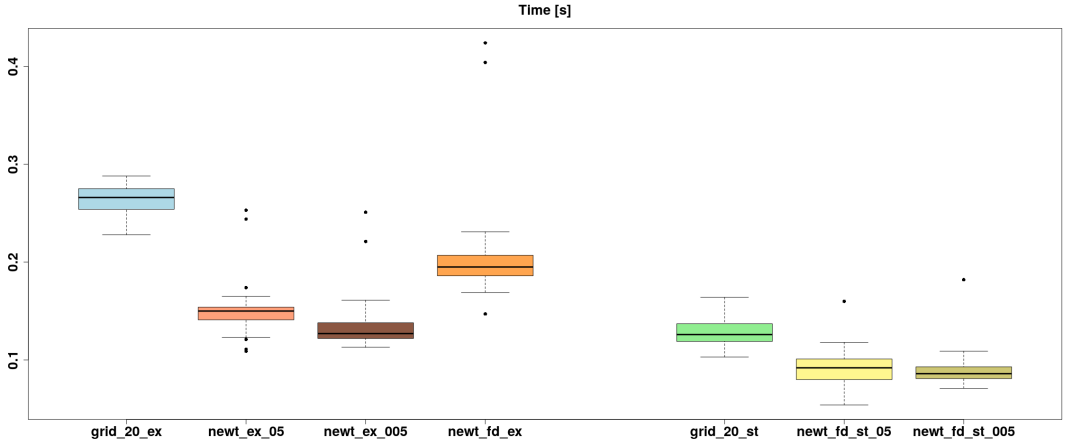
(f) `newt_ex_0.005`

Figure 5.7: Test 5 (C mesh): comparison among the different output fields

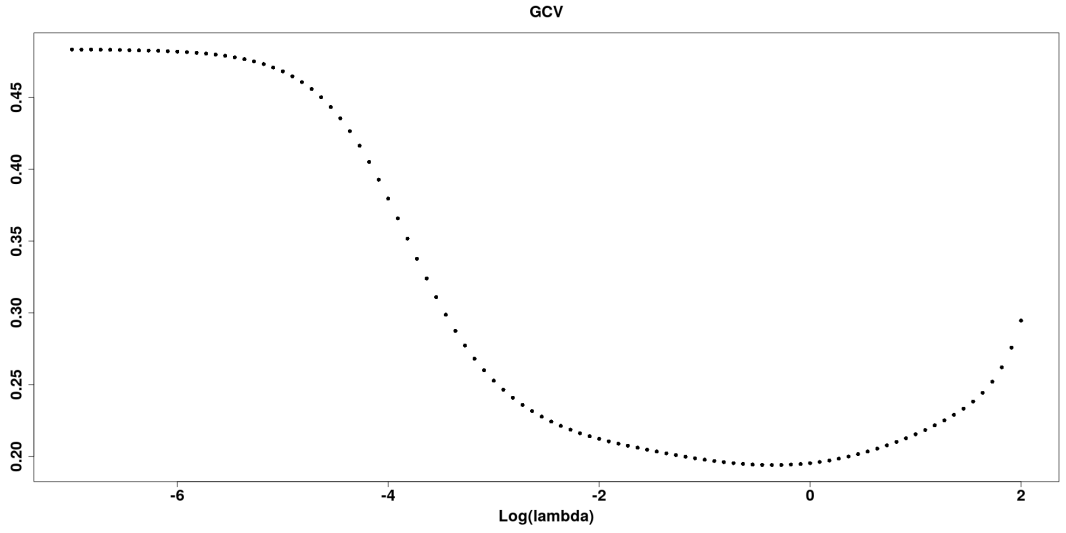
Mesh C - Test 6 [264 nodes - 171 observations - no covariates - point locations not at nodes] In this test we consider the same setting as the previous one, but without covariates. We can observe in Figure 5.8 that Newton's methods have better performances than the grid evaluation counterpart with 20 values of λ, in terms of RMSE. In this case, a tolerance of 0.005 leads to slightly better RMSE results, with less computational time than the corresponding 20-values grid.



(a) Method vs RMSE



(b) Method vs Time



(c) GCV as function of λ

Figure 5.8: Test 6 (mesh C), 264 nodes - 171 observations - no covariates - point locations not at nodes

Method	Optimal λ	GCV	Average time
grid_20_ex	4.28e-01	1.92e-01	2.66e-01
newt_ex_0.05	2.26e-01	1.94e-01	1.50e-01
newt_ex_0.005	3.12e-01	1.92e-01	1.27e-01
newt_fd_ex	2.26e-01	1.94e-01	1.95e-01
grid_20_st	4.28e-01	1.93e-01	1.26e-01
newt_fd_st_0.05	2.28e-01	1.95e-01	9.20e-02
newt_fd_st_0.005	3.20e-01	1.94e-01	8.60e-02

5.1.3 PDE case

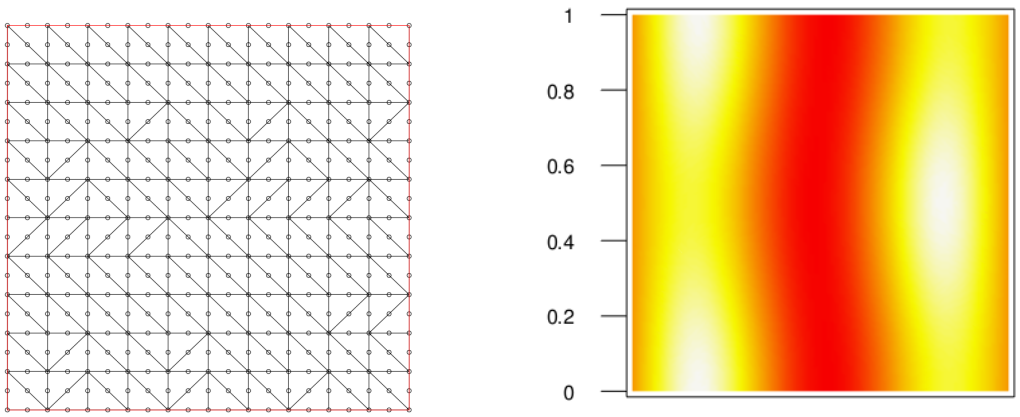
The true function used in this test is:

$$f(x, y) = \sin(2\pi x)\cos(2\pi y) + 4\sin(3\pi x).$$

We do not consider the covariates and we fix the observation locations at the mesh nodes. Nevertheless, recalling the model (1.4), we introduce the following term:

$$K = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$$

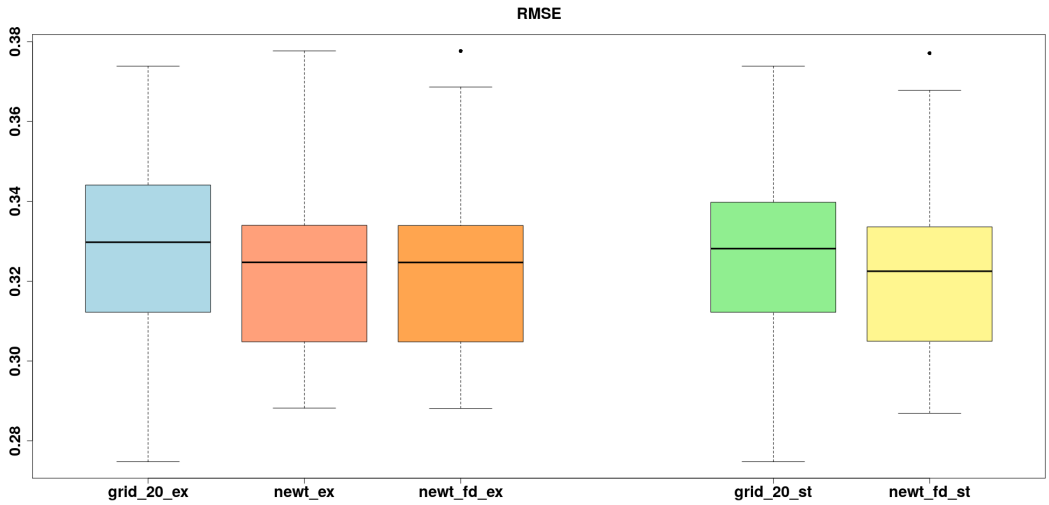
which will be employed as anisotropy matrix.
The mesh is the following square mesh and exact function f:



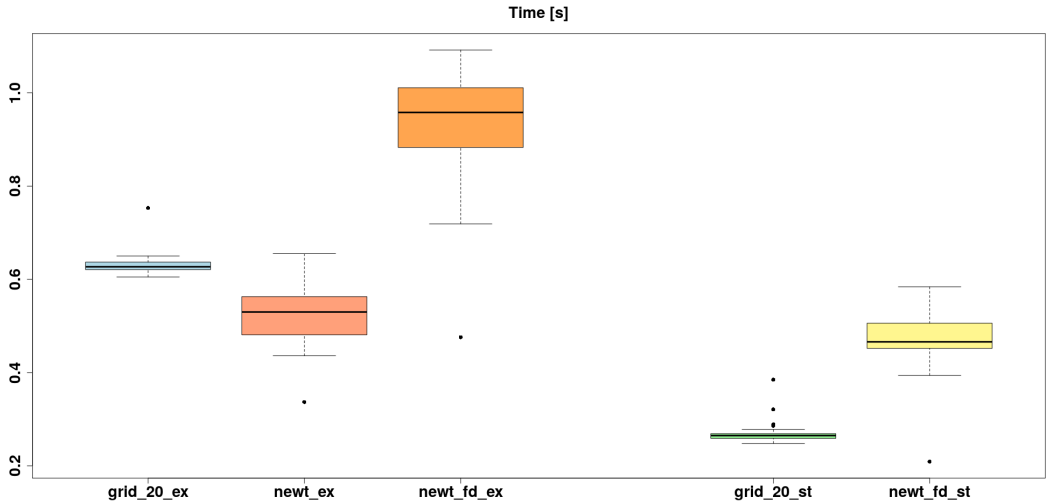
(a) Computational mesh and sample locations. **(b)** Representation of the not perturbed function f.

Figure 5.9: Mesh PDE: mesh nodes 441 - observations 441, locations are nodes

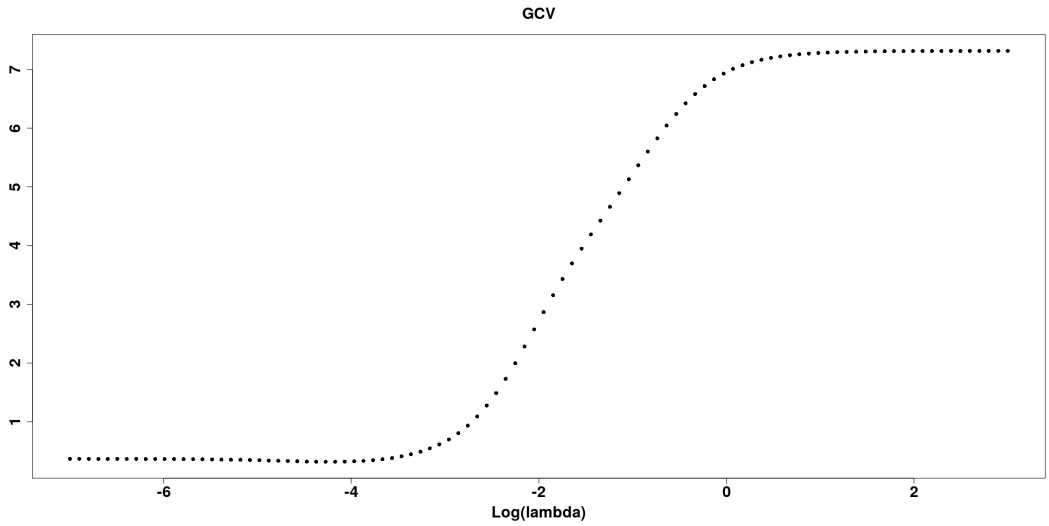
Mesh PDE - Test 7 [441 nodes - 441 observations - point location at nodes]
We can observe in Figure 5.10 that Newton's methods have better performances than grid evaluation strategies with 20 values of λ, in terms of RMSE. Namely, in this case, a tolerance of 0.05 leads to better RMSE results, without the necessity of choosing smaller values. In this situation we see that also the computational times are comparable, suggesting that the Newton's methods represent globally a better choice in this case. In this test we do not observe much difference between the exact GCV computational time and the stochastic, since the nodes are small in number. Moreover, the representation of the estimated field $\hat{f}$ are similar to each others, since the values of λ are about the same for all the cases see Appendix D for a reference.



(a) Method vs RMSE



(b) Method vs Time



(c) GCV as function of λ

Figure 5.10: Test 7 (mesh PDE), 441 nodes - 441 observations - point location at nodes

Method	Optimal λ	GCV	Average time
grid_20_ex	6.95e-05	3.077e-01	0.627
newt_ex	7.62e-05	3.071e-01	0.530
newt_fd_ex	7.63e-05	3.071e-01	0.958
grid_20_st	6.95e-05	3.084e-01	0.383
newt_fd_st	7.53e-05	3.079e-01	0.288

5.1.4 A monotone increasing GCV function

In this test we show the behaviour of the optimization methods in case of a monotone increasing GCV function. The mesh is divided into 7 areal regions. Hence, we consider an areal smoothing, with the following exact function f (contained in the file `quasicircle2Dareal.RData`) whose evaluation on the areas produces values:

$$\mathbf{f} = [20.70822, 16.77969, 24.87815, 19.26803, 28.60480, 19.14191, 19.06811]^T$$

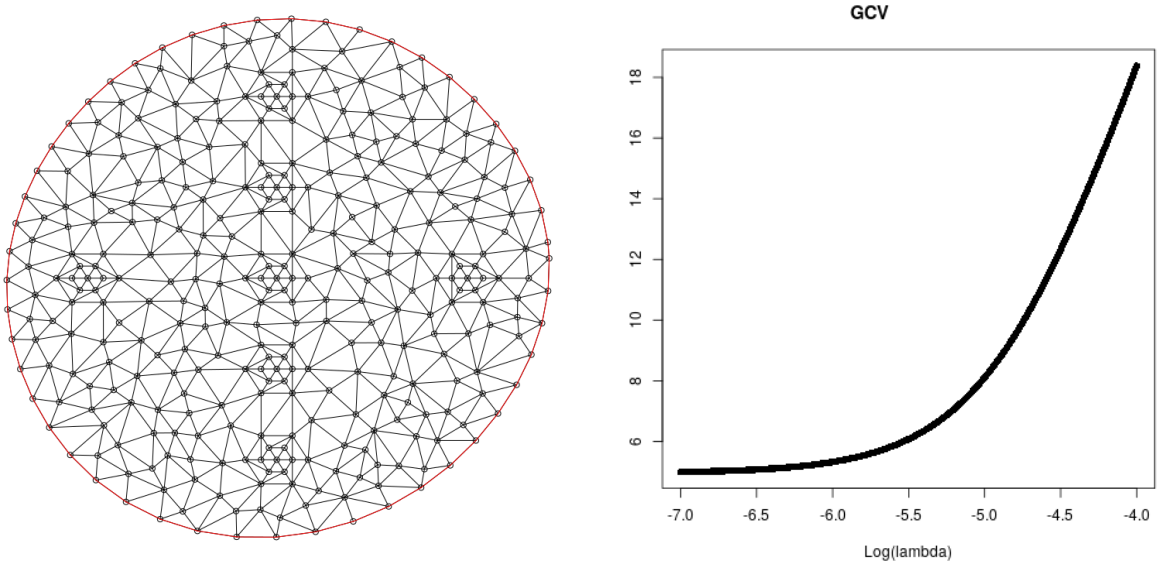
Moreover, the following PDE parameters are used:

$$K(x,y) = \begin{bmatrix} y^2 + k_1 x^2 + k_2(R^2 - x^2 - y^2) & (k_1 - 1)xy \\ (k_1 - 1)xy & x^2 + k_1 y^2 + k_2(R^2 - x^2 - y^2) \end{bmatrix},$$

where $R = 2.8$, $K_1 = 0.1$, $K_2 = 0.2$ and $\beta = 0.5$;

$$\mathbf{b}(x,y) = 10\beta[x,y]^T.$$

The forcing term $\mathbf{u}$ is equal to 100 if $x^2 + y^2 < 1$ and 0 otherwise. In conclusion the boundary conditions are fully Dirichlet, imposing that $f = 5$ on the boundary. The mesh used is represented in Figure 5.11a. In this case the GCV function has the shape in Figure 5.11b.



(a) Computational mesh.

(b) GCV is monotone increasing.

Figure 5.11: Principal structures of the example

This means that in grid evaluation the best value is the smallest possible value. As we expected, all the Newton's methods available in this case (GCV stochastic with finite differences and GCV exact with finite differences) print the same output:

```
1 Probably monotone increasing GCV function
```

showing that the methods are able to recognize this kind of shapes.
Clearly, in this case is is not useful to compare the methods with the grid evaluations, since we can choose an arbitrary small value of λ and obtain a smaller and smaller RMSE. The aim of the Newton's methods in this case is to identify the shape of the GCV and thus suggest to use a very small λ.

5.2 2.5D smoothing

To test the 2.5D smoothing we chose the following function f:

$$f(x,y,z) = \sin[4(0.5\sin(\theta)\exp(-\sin(\theta)^2)+1)\theta]\cos[2(0.5\cos(\phi)\exp(-\cos(\phi)^2)+1)\phi]$$

where

$$\phi = \arcsin(\frac{y}{\sqrt{x^2+y^2}})$$
$$\theta = \arccos(\frac{z}{\sqrt{x^2+y^2+z^2}})$$

The mesh is represented together with the exact function f in Figure 5.12, the observation locations coincide with some nodes

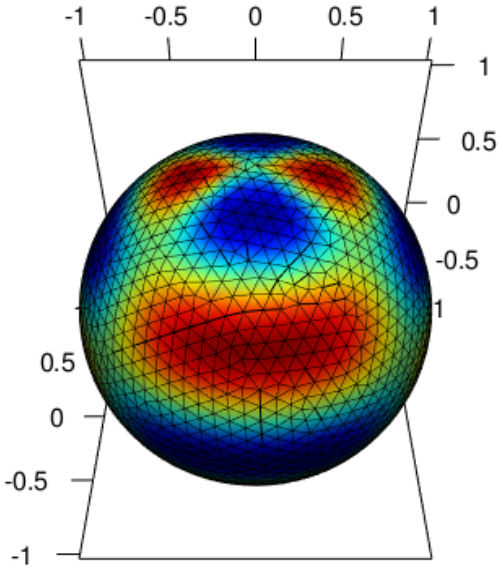


Figure 5.12: Computational mesh and the f field

We consider one covariate:

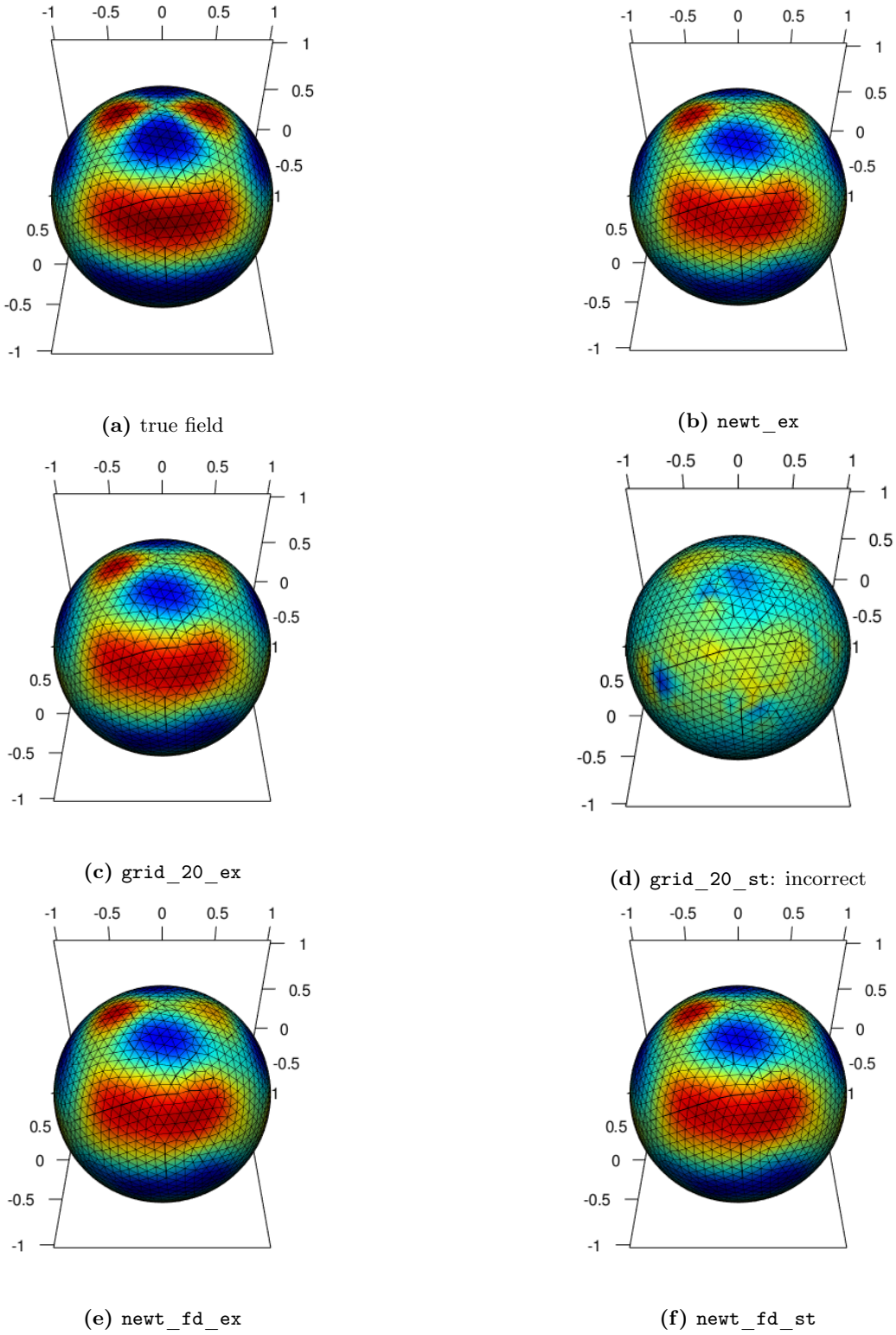
$$\begin{cases} w_{1i} \sim \mathcal{U}([-1,1]) \quad i = 1,\dots,n \\ \beta = 1 \end{cases}$$

The results are then resumed in the next Tables: for the grid evaluation, since in general for the 2.5D and 3D cases the range of λ could change, we used the following vector of 20 values:

```
lambda = 10^seq(-8,3,length.out=20)
```

2.5D Sphere - Test 8 [2'113 nodes - 500 observations - 1 covariate - locations not at nodes] As we can see in Figure 5.14, we notice that the λ found is approximately the minimum for the GCV in the plot. The grid stochastic method has more variability than the corresponding Newton's stochastic counterpart, and even a worse RMSE: it takes abut 2 seconds less, but for what concerns the performances the Newton's method is better. Indeed, we notice that all the methods, apart from the `grid_20_st`, have a similar RMSE variability (no outliers), confirming the stability of the Newton's finite differences method with stochastic GCV. The variability of the method `grid_20_st` is problematic, since if we try to plot the estimated field $\hat{f}$ for the case of an outlier of the RMSE boxplot, $\lambda = 1.0\text{e}{-08}$, we obtain a completely wrong estimation, as we can see in Figure 5.13d. Thus, the higher robustness of the Newton's methods is a big

improvement. About the exact evaluations, which take much more time than the stochastic ones, we notice that the Newton's exact strategy has the best performances, since it takes much less time, but guarantees approximately the same RMSE than the grid evaluation (see Appendix D for a reference). Again we stress the fact that the Newton's method performs better than the finite differences method, since the former is optimized in the computations.



(a) true field

(b) `newt_ex`

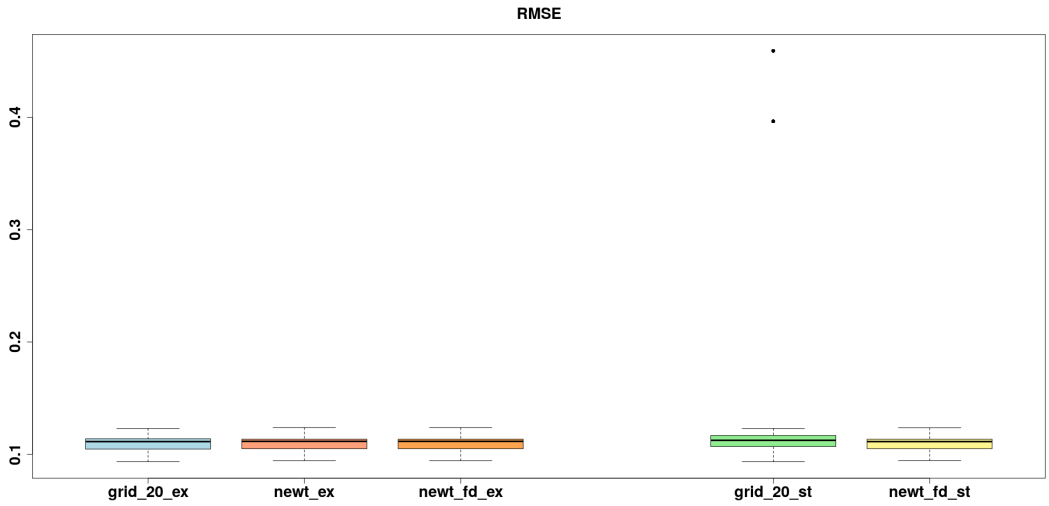
(c) `grid_20_ex`

(d) `grid_20_st`: incorrect

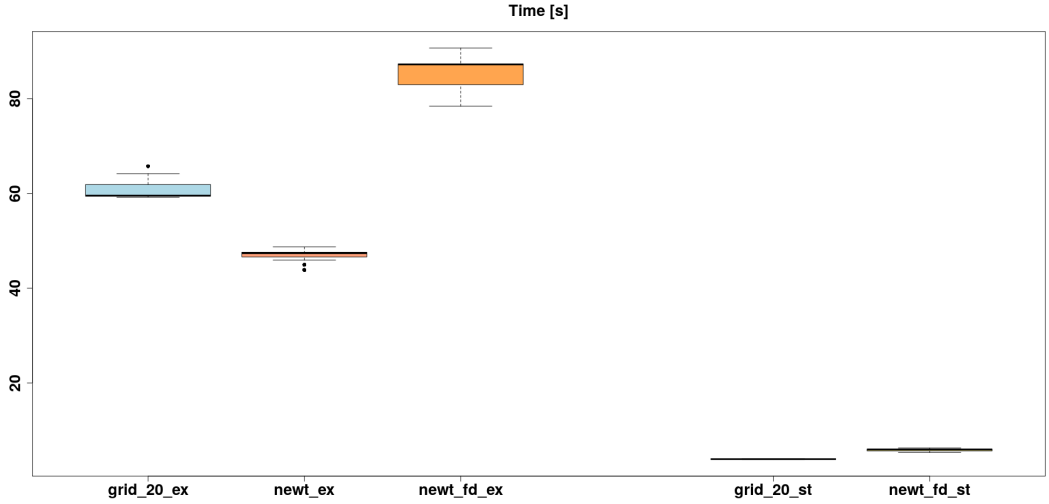
(e) `newt_fd_ex`

(f) `newt_fd_st`

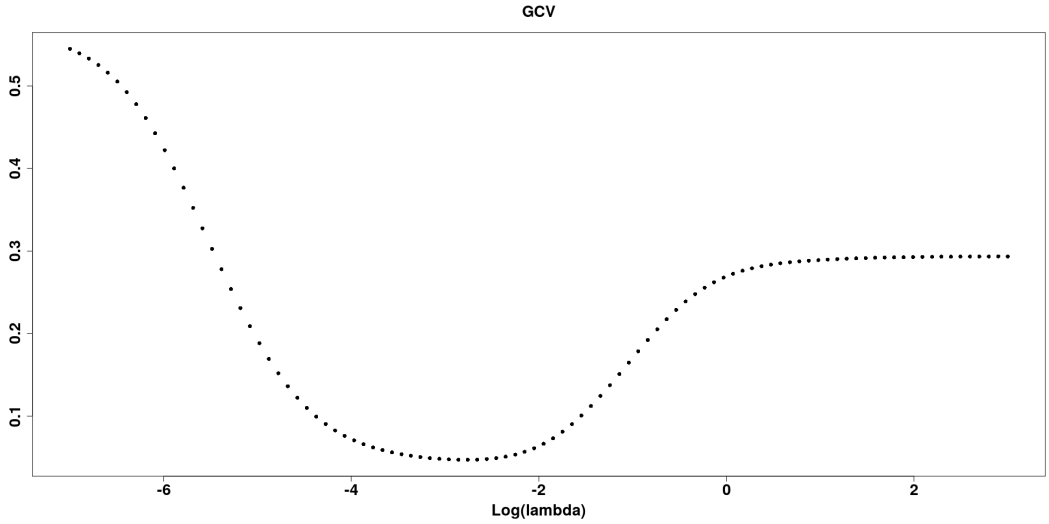
Figure 5.13: Test 8 (2.5D Sphere): comparison among the different output fields



(a) Method vs RMSE



(b) Method vs Time



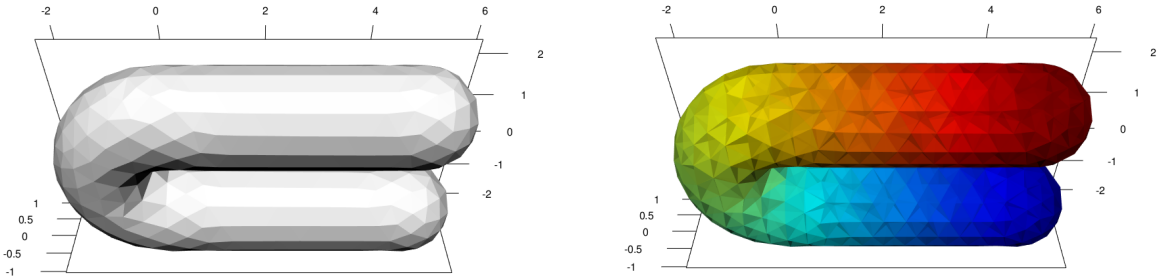
(c) GCV as function of λ

Figure 5.14: Test 8 (2.5D sphere), 2'113 nodes - 500 observations - 1 covariate - locations not at nodes]

Method	Optimal λ	GCV	Average time
grid_20_ex	1.6e-03	4.8e-02	59.5
newt_ex	1.8e-03	4.8e-02	50.76
newt_fd_ex	1.8e-03	4.8e-02	93.4
grid_20_st	1.6e-03	4.8e-02	3.9
newt_fd_st	1.8e-03	4.8e-02	6.5

5.3 3D smoothing

As 3D test we consider the 3D corresponding to the mesh C, as in Figure 5.15a. In this case, the observation locations coincide with all the mesh nodes (i.e. $n = N$). As exact function f we consider the same as in Section 5.1.2, in 3D version, which is represented in Figure 5.15b:



(a) Computational mesh.

(b) The exact field f.

Figure 5.15: Principal structures of the example

We add two covariates, which are the following:

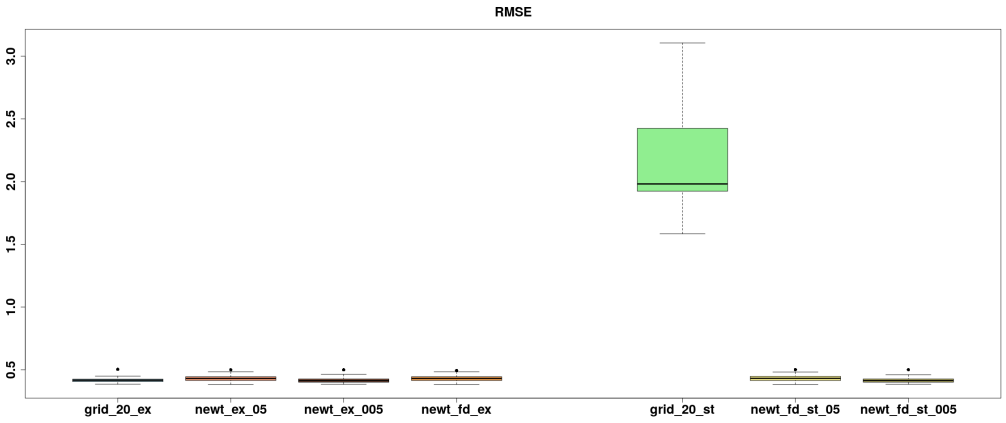
$$\begin{cases} w_{1i} = sin(x_i) & i = 1, \dots, n \\ w_{2i} \sim \mathcal{N}(\mu = 1, \sigma = 2) & i = 1, \dots, n \end{cases}$$

As in the 2.5D case, we changed the grid evaluation: we consider the following 20-values vector of λ for the grid evaluation:

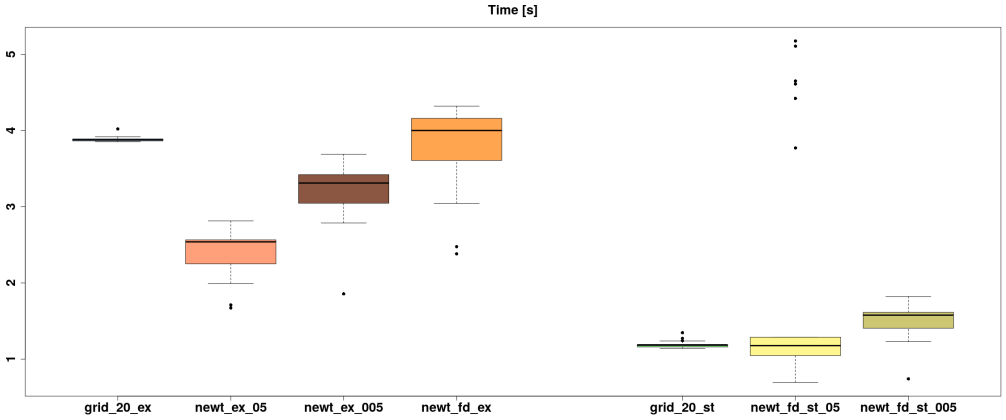
```
1 lambda = 10^seq(-7,3,length.out=20)
```

The results are then summarized in the following Tables.

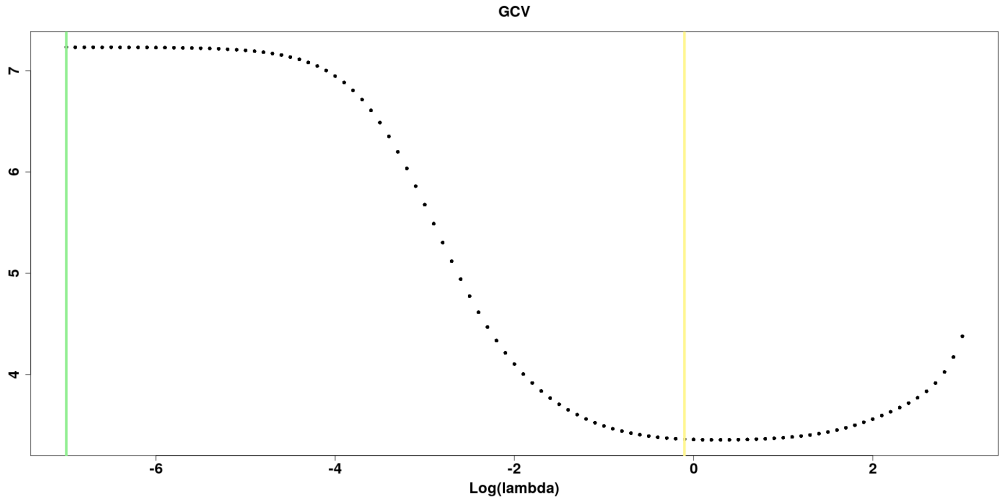
3D mesh C - Test 9 [724 nodes - 724 observations - 2 covariates - locations not at nodes] The grid stochastic strategy has more variability than the corresponding Newton's stochastic method, and even a worse RMSE: it takes abut 0.5 seconds less, but for what concerns the performances the Newton stochastic is better. Indeed, it results to be more stable, since the stochastic GCV is not reliable form very small values of λ (we obtain also a very different plot of $\hat{f}$ from the original one, as we see in the comparison of the plots, Figure 5.17d). Also in Figure 5.16c we observe that the average λ chosen in the stochastic grid evaluation method is very different from the one chosen by the Newton's method, which is also the nearest to the minimum of the GCV function. About the exact evaluations, which take more time than the stochastic ones, we notice that the Newton's exact method has the best performances, since it takes much less time but guarantees approximately the same RMSE than the grid evaluation. Again we stress the fact that the Newton's method performs better than the finite differences since the former is optimized in the computations.



(a) Method vs RMSE



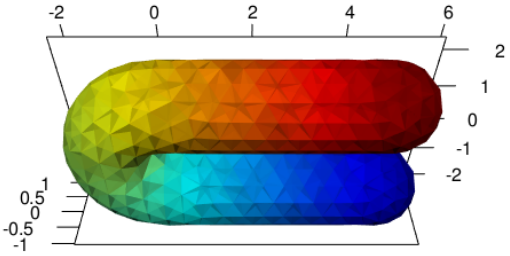
(b) Method vs Time



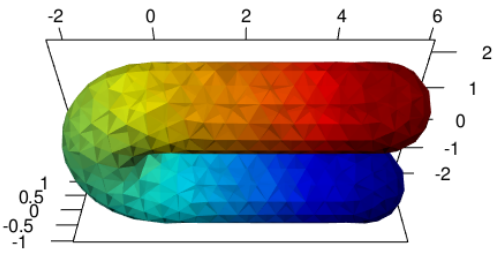
(c) GCV as function of λ: the colored lines represent the optimal λ in the stochastic grid and Newton's method.

Figure 5.16: Test 9 (3D mesh C), 724 nodes - 724 observations - 2 covariates - locations not at nodes

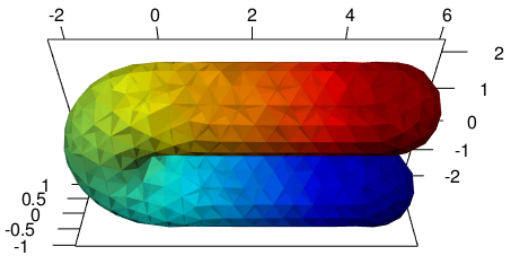
Method	Optimal λ	GCV	Average time
grid_20_ex	2.34e+00	3.10e+00	3.9
newt_ex_0.05	0.85e+00	3.15e+00	2.5
newt_ex_0.005	2.11e+00	3.11e+00	3.3
newt_fd_ex	0.85e+00	3.15e+00	4.0
grid_20_st	1.00e-07	1.54e-03	1.2
newt_fd_st_0.05	0.79e+00	3.15e+00	1.2
newt_fd_st_0.005	2.1e+00	3.11e+00	1.6



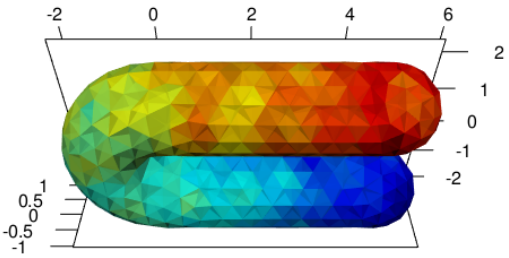
(a) true field



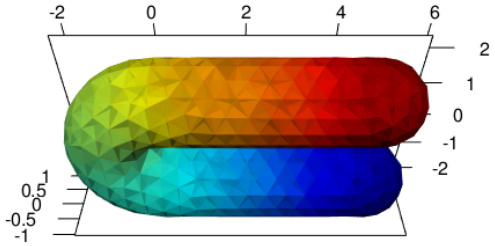
(b) `newt_fd_st`



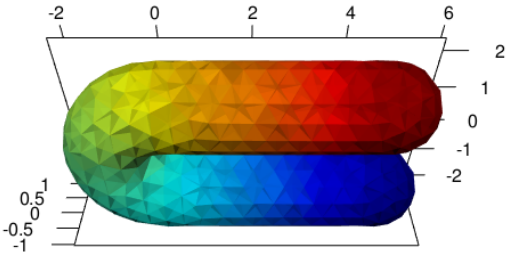
(c) `grid_20_ex`



(d) `grid_20_st`: incorrect



(e) `newt_ex_0.05`



(f) `newt_ex_0.005`

Figure 5.17: Test 9 (3D mesh C): comparison among the different output fields

5.4 Performance comparison with the previous version of `fdaPDE`

The aim of this Section is to compare the new GCV evaluation method (grid evaluation) to the analogous method already implemented in the previous version of the library [5]. Firstly, we analyzed the test in Section 5.1.1, increasing the number of nodes (coinciding with the locations), and we compared the performances for the exact GCV computation and the stochastic GCV computation. In the following Table we summarize the results, as the number of nodes increases. We choose to compute the GCV at the value $\lambda = 1e{-}05$, repeating the tests twenty times and computing the mean time for each case. The entries indicated by N.A. mean that the computation time was too high to be a viable option. Namely, for the exact GCV computation it is useless to go beyond 6'000 nodes, since the time is more than 10 minutes for just one evaluation.

Nodes	Stoc. old code	Stoc. new code	Exact old code	Exact new code
1'681	0.094	0.088	2.6	2.56
3'281	0.24	0.209	29.5	33.5
6'597	0.53	0.484	300	320
15'741	1.59	1.55	N.A.	N.A.
30'971	3.71	3.62	N.A.	N.A.
39'125	4.82	4.75	N.A.	N.A.

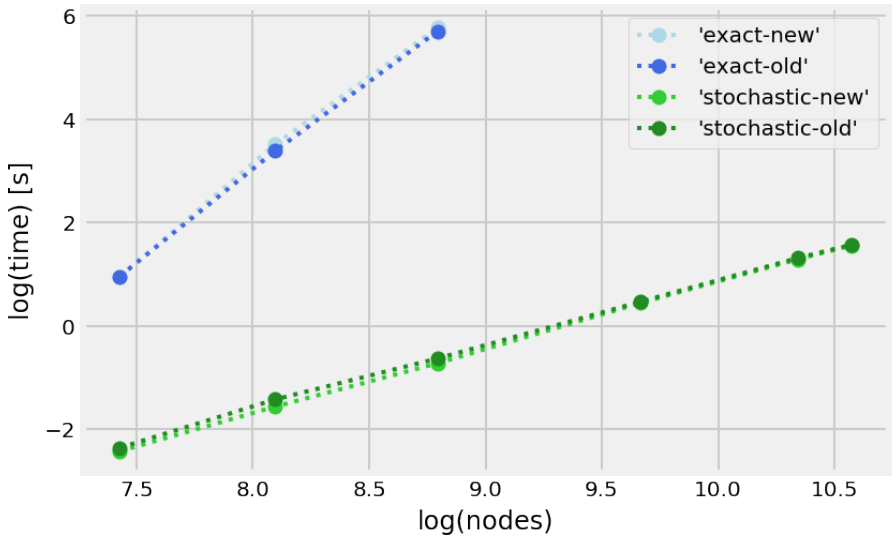


Figure 5.18: log-log plot of nodes vs times in squared mesh

As we can also see in Figure 5.18, we notice that the computational costs for the exact GCV are slightly higher than in the previous version of the code: this is because now the computation of the $\hat{\mathbf{z}}$ is done starting from the matrix S (see eq. 3.2), thus to compute also $\hat{\boldsymbol{\beta}}$'s we need to call the function `apply` at the end of the grid evaluation, which was not performed in the previous version of the library. This is done only for the λ corresponding to the minimum GCV, but in case of only one value this is always executed, adding the computational cost of the solution of the system 1.8. Actually, this additional cost in the grid evaluation is performed only once, then the more λ's are evaluated the less the time of this last call to `apply` has an impact on the evaluations.

In the case of the stochastic GCV, instead, since the method `apply` is already used to compute $\hat{\mathbf{z}}$, we do not need to call it again in the end of the grid evaluation of a single value: thus the computational time is even a little better than the previous code. Since the most important methods are the stochastic ones, due to their reduced computational costs (see [3]), we consider acceptable this result, having improved a bit the computational time of this case.

We then considered the test of Section 5.1.2, increasing the number of observations (whose locations do not coincide with the nodes) and nodes in the following way, keeping the ratio between nodes and observations between 50% and 60%.[1] We used a value of $\lambda = 0.001$. We see the results in the following Table.

Obs./Nodes	Stoc.old code	Stoc.new code	Exact old code	Exact new code
171/264	0.024	0.021	0.033	0.035
399/754	0.071	0.066	0.306	0.393
2789/5703	1.103	0.872	175	188
6938/10423	1.9	2.2	N.A.	N.A.

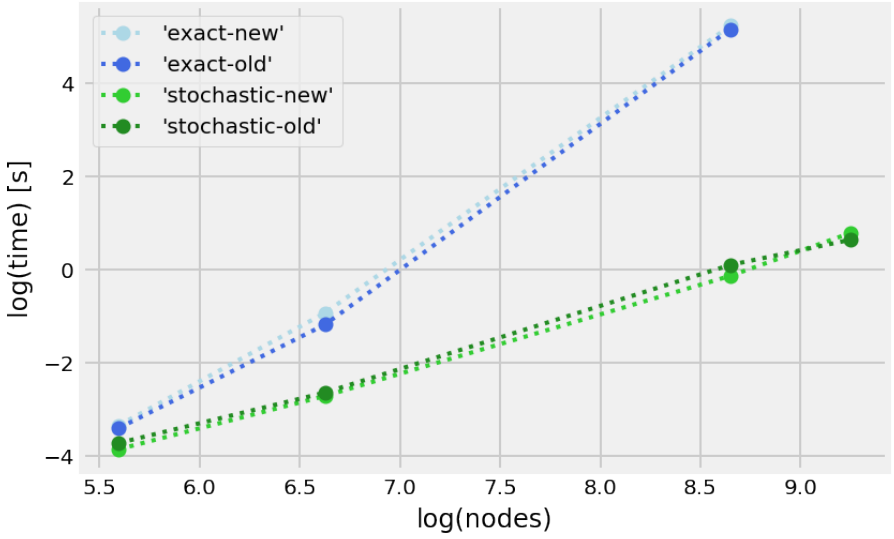


Figure 5.19: log-log plot of nodes vs times in horseshoe mesh

Observing the plots in Figure 5.19, we notice that the exact GCV computations take slightly more time in our new version of the code, due the last application of the method `apply`, which is nevertheless of less impact if we evaluate a grid of more than one λ, whereas for the stochastic evaluation we see for most of the cases a slight improvement in the computational time, as in the previous case. It is important to stress that much of the time consumption is spent, in stochastic methods at least, in building the output. So small fluctuations between the newer and former version are reasonable. E.g. note that, of the 2.2 second used for the 10423 nodes case, almost 0.7 are spent in building the richer output we have described in Section 4.7.

In conclusion, since the best improvements are seen in the stochastic GCV computation and this is the most relevant and used case, we incite to preserve as default way of the GCV computation the stochastic one.

[1] Again we denote by N.A. the times for the computation of exact GCV which are not affordable, and thus useless, for the user, because too long. Moreover we denote with **Obs.** the number of observations.

5.5 Conclusions from the simulations

From the simulations performed we can summarize the following results:

- First of all we point out the main advantages of the Newton's methods compared to the grid evaluation.

  The grid evaluation method is a semi-automatic method, since it strongly depends on the grid of λ which are given as input by the user. This could lead to two different orders of problems: if the user is not an expert of the library, he could try to use a grid of λ which does not contain - or even is very far from - the real minimum of the GCV. In this way the user could be misleadingly convinced that the best λ is present in his grid and thus commit large errors. Indeed we stress again that the choice of the smoothing parameter is crucial to get a good smoothing performance. If the user is an expert one, he should nevertheless make different tries of various grids of λ in order to find a good one.

  On the other hand, our newly implemented Newton's methods are completely automatic: even without taking as input any initial λ, they are able to find the minimum of the GCV function under a tolerance, which is user defined: no grids are required and the user does not need to do many tries to find the best λ.

- Moreover, for what concerns the perfomances of the Newton's methods, we distinguish the two cases, i.e. the exact GCV computation and the stochastic computation.

  - About the exact GCV computations, we have observed that the Newton's method with exact evaluation of derivatives is very effective. Indeed it allows to find an optimal λ with a small RMSE, compared to a grid with a sufficient number of λ to be evaluated. Namely, Newton's method is faster and reaches a value of RMSE comparable or even smaller than the grid evaluation (see, e.g., tests in Sections 5.1.1 and 5.1.3). Compared to the Newton's method with finite differences, the exact derivative evaluation performs better in terms of time computation, thus we suggest to use, in case of exact GCV, the Newton's method with exact derivatives instead of the corresponding grid evaluation one.
  - For what concerns the stochastic GCV we noticed that, in general, it takes more time than grid evaluation, but in most of the cases it has a better performance in terms of RMSE (see, e.g. the test in Section 5.1.3). Since in general the user does not know how many values of λ's are needed in a grid to get a good RMSE, the automatic iterative Newton's method is surely a better and simpler way to choose the best smoothing parameter: in one shot it gives a good result without any λ input from the user.

- Another important issue concerns the stochastic GCV evaluation: as we have noticed in tests 5.1.2, 5.2 and 5.3, the grid evaluation of stochastic GCV, which is the actual default method of the library, is not reliable, since it gives completely wrong results in many non pathological cases. On the other hand we have observed in the same tests that the finite differences Newton's method with stochastic GCV is much more robust, in the sense that most of the times it gives the correct estimation of the best smoothing parameter λ, even when the corresponding grid method fails. Therefore, this is another strong reason to prefer the Newton's method to the grid one when evaluating the stochastic GCV.

- For what concerns the tolerance to be used in the Newton's method as stopping criterion, we noticed that in many cases a value of 0.05 is enough to obtain good a good performance in terms of time needed and RMSE (see, e.g., the tests in Section 5.1.1 and 5.1.3), thus we decide to set this value as default. The user can decrease it, using, for example, 0.005, which we have seen makes the result even better in some cases (see again the tests in Section 5.3).

- In conclusion, in the test in Section 5.1.4, we show that the Newton's methods are able to recognize strange behaviours of the GCV functions, in particular the monotone increasing behaviour is identified and referred to the user with an output message.

Chapter 6

Package Installation

The source code described in this report can be found and downloaded from:
`https://github.com/GMeretti/PACS_merettipoiatti`.

There are many ways to install an R source package, but not all of them work for Windows. Here we suggest two that do the job (we recall that `R 3.5.0` or newer is needed for the installation):

- Method 1: download the .zip file from GitHub and unzip it on your local PC. On the `R` console, letting `.../sd` the path to the super-directory where `PACS_merettipoiatti − master.zip` has been extracted, type:

  ```
  install.packages(".../sd/PACS_merettipoiatti-master", type='source',
      repos=NULL)
  ```

  which surely works on both Windows and Linux. If some issue arises, in particular on Windows, it might be due to the name, so rename `PACS_merettipoiatti − master`, the folder containing only the library (`src`, etc.), in `fdaPDE` and move it for example in the folder `Documents`: this might solve the problem of finding the directory, after changing the folder path in the installation.

- Method 2: From the terminal, run:

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

  which works on Linux with this syntax.
  In this case:
  1) `path name of the R library tree` represents the path where you want the library to be installed (in general it is in the directory `R`);
  2) `path name of the package to be installed` represents the path of the package folder;
  Since this manual method is more involved, we suggest to make use of the first one to install the library.

When installing the library you need to install auxiliary packages, which are needed for the complete installation:

- **For Ubuntu users**: To install `R` on Ubuntu, see, for instance, [18]. In general the libraries `rgl`, `plot3d`, `geometry` and `plot3Drgl` are needed. After the installation of these libraries, the installation of the library `fdaPDE` should be completed without other interruptions.

- **For Windows users** the installation requires different step on the basis of the `R` version:

  - **For R 4.0.2**: you need to install Rtools40 from https://cran.r-project.org/bin/windows/Rtools/ and follow the instructions described there to complete the installation. This package is necessary to install any R package. Then, before installing the library fdaPDE, you are asked to install the libraries RcppEigen, rgl, plot3d, geometry and plot3Drgl. After these installations, the package should work properly.
  - **For R less recent versions**: you need to install Rtools35 from https://cran.r-project.org/bin/windows/Rtools/history.html. This version is older than the one previously described. In this case it could happen that no other extra libraries are needed to be installed, or you are asked to install the same ones as in the more recent version of R already explained, and the installation of the library fdaPDE should work properly.

- **For macOS users**: before installing the library you have to install the libraries RcppEigen, rgl, plot3d, geometry and plot3Drgl.
  After their installation, the command install.packages should work without any problems.

We did not find any other possible error in Linux, but according to the report [4], there could occur the following, while installing the library rgl:

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

or

```
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
```

which could be solved by installing

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

A similar error could happen for macOS users, again while installing the library rgl, due to the absence of X11.
You can solve the issue by installing the library XQuartz from https://www.xquartz.org/.

For Windows users we recall that if Rtools40 seems not to work properly, you need to run the following R command:

```
writeLines('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"', con = "~/.Renviron")
```

and then try to run

```
Sys.which("make")
```

which should give as a result, if all is working correctly:

```
"C:\\rtools40\\usr\\bin\\make.exe"
```

as well described at https://cran.r-project.org/bin/windows/Rtools/.

6.1 Installation checks

In order to check if the installation could be correctly performed on different machines, we checked successfully the following cases, with the corresponding `R` version used:

- Ubuntu 18.04, `R` 3.5.3
- Ubuntu 20.04, `R` 3.6.3 and `R` 4.0.2
- macOS 10.15.4, `R` 4.0.2
- Win10, `R` 3.5.0 (necessary also library `shiny`), `R` 3.5.3, `R` 3.6.3 and `R` 4.0.2

Furthermore, we also virtually checked, by means of the `R` package `rhub`, the following machines:

- macOS 10.13.6 High Sierra, R-release
- macOS 10.13.6 High Sierra, R-release, CRAN's setup

with successful results.

6.2 Testing example

To test the successful installation of the package, we would suggest to run the following script: `smooth.FEM.2D.tests.R` in the sub-directory `tests` either from the terminal using

```
Rscript <smooth.FEM.2D.tests.R>
```

or directly from `R`, once in the proper directory. For example you can use

```
source("smooth.FEM.2D.tests.R")
```

This script contains a list of tests of 2D smoothing which covers most of the package functionalities. You can also use, with the same commands, the test cases `smooth.FEM.2.5D.tests.R`, containing the 2.5D smoothing tests, and `smooth.FEM.3D.tests.R`, containing the 3D smoothing ones (be careful that the first test in the 3D smoothing takes a lot of time to run, as it is pointed out in a comment before the test).

Each test is composed of the setting of the mesh and the data and then a call to the main function. For example, looking at the first test in `smooth.FEM.2D.tests.R` we find the function calls:

```
#### Test 1.1: Without GCV
output_CPP<-smooth.FEM(observations=data, FEMbasis=FEMbasis, lambda=lambda)
image(output_CPP$fit.FEM)

#### Test 1.2: grid with exact GCV
output_CPP<-smooth.FEM(observations=data, FEMbasis=FEMbasis, lambda=lambda,
    lambda.selection.criterion='grid', DOF.evaluation='exact',
    lambda.selection.lossfunction='GCV')
plot(log10(lambda), output_CPP$optimization$GCV_vector)
image(FEM(output_CPP$fit.FEM$coeff,FEMbasis))

#### Test 1.3: grid with stochastic GCV
output_CPP<-smooth.FEM(observations=data, FEMbasis=FEMbasis, lambda=lambda,
    lambda.selection.criterion='grid', DOF.evaluation='stochastic',
    lambda.selection.lossfunction='GCV')
plot(log10(lambda), output_CPP$optimization$GCV_vector)
image(FEM(output_CPP$fit.FEM$coeff,FEMbasis))
```

```
### Test 1.4: Newton exact method with exact GCV, default initial lambda and
    tolerance
output_CPP<-smooth.FEM(observations=data, FEMbasis=FEMbasis,
    lambda.selection.criterion='newton', DOF.evaluation='exact',
    lambda.selection.lossfunction='GCV')

### Test 1.5: Newton_fd method with exact GCV, default initial lambda and tolerance
output_CPP<-smooth.FEM(observations=data, FEMbasis=FEMbasis,
    lambda.selection.criterion='newton_fd', DOF.evaluation='exact',
    lambda.selection.lossfunction='GCV')

### Test 1.6: Newton_fd method with stochastic GCV, default initial lambda and
    tolerance
output_CPP<-smooth.FEM(observations=data, FEMbasis=FEMbasis,
    lambda.selection.criterion='newton_fd', DOF.evaluation='stochastic',
    lambda.selection.lossfunction='GCV')
```

Each call tests a different case implemented, with the possible addition of extra parameters. In particular, in Newton's cases, you can also specify parameter `lambda.optimization.tolerance` to set the tolerance for the iterative optimization method, defaulted to 0.05, and, for the stochastic case, set `DOF.stochastic.realizations` or the `DOF.stochastic.seed` (see Section 4.1.1 for insight), which is used for the computation of the stochastic Degrees Of Freedom used in the GCV, and randomly set if not explicitly set by the user. The expected results of the presented test can be found in the test Section 5.1.1.
All the other test cases are organized in a similar way, and we invite the user to look at the `R` scripts to see them in detail.

Chapter 7

Guidelines for Future Code Maintenance

7.1 Documenting C++ code

The code written in our C++ classes (directory src/Lambda_Optimizer) is fully able to support Doxygen documentation. In order to see visualize it you may go to the src directory and write on terminal:

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

Where <config-file> will be the name of the configuration file that can be edited to customize the output (standard name Doxyfile). The editing *must* contain the following alterations:

```
RECURSIVE = YES
GENERATE_HTML= YES
```

otherwise you won't be able to parse all the sub-directories of folder src.
We also suggest the use of the following changes in order to have complete documentation:

```
PROJECT_NAME = "fdaPDE"
TAB_SIZE = 8
EXTRACT_ ALL = YES
EXTRACT_PRIVATE = YES
EXTRACT_PACKAGE = YES
EXTRACT_STATIC = YES
EXTRACT_LOCAL_METHODS = YES
EXTRACT_ANON_NSPACES = YES
```

You may also modify the location of <config-file> and simply add INPUT = ../path_to_src. Finally, to run Doxygen type on terminal:

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

To see the final result go in directory html, open file index.html and browse across the features.

7.2 Documenting R code

All documentation in R packages is stored in a sub-directory called man (at the same level of src and R to be clear), consisting of files with .Rd extension - that obviously stays for R-documentation. Every time something is changed either in directories R or data, all the documentation has to

be rewritten in order to be up to date. Note that this procedure, despite being a prerequisite for a CRAN release, is not relevant for compilation issues. Thus, it is not strictly necessary to redo this operation while developing `fdaPDE`. Instead, it is rather something that should be performed when commenting the code, possibly on a stable version.

A clever trick to let the documentation to be produced automatically is to use R package Roxygen2, see [20] for further details. If you are more familiar with Doxygen just consider that Roxygen2 is pretty much analogous, just the commenting style is a bit different. We leave as example to look at function `smooth.FEM` in `R/smoothing.R` in order to understand how user-friendly it is to write in Roxygen. Not being our project mainly focused on deep `R` integration, we would suggest the interested reader to look at `http://r-pkgs.had.co.nz/man.html` if further insight of package documentation and Roxygen2 syntax will be required.

To finally produce documentation in `RStudio`, set your current directory in the package folder and call Roxygen2:

```
setwd("...\PACS_merettipoiatti-master")
roxygen2::roxygenise()
```

This alone will produce both `man` directory and the whole documentation written inside, exactly with the format and extension described above.

7.3 Future developments

During the development of the code we found some possible interesting extensions of the library, which we propose here as possible future object of study.

In particular, it could be useful to introduce a different criterion from the GCV function for the estimation of the best smoothing parameter λ, like the k-fold Cross Validation.
To this aim, we have already left the declaration of a possible extension in this direction in file `src/Lambda_Optimization/Lambda_Optimizer.h`: the creation of a new class of evaluation methods is an immediate extension and then it can be used in the optimization methods already implemented without difficulties.

For what concerns the optimization techniques used to minimize the GCV function, or another similar function, we have observed from the tests performed that Newton's methods are very effective, thus we have not explored other strategies. Nevertheless, if necessary, in the future, other optimization methods could be added, by simply inheriting from the already present father class `Opt_methods`.

We performed the optimization procedure only for the spatial case, but the same ideas could be extended to the spatio-temporal case and also to the FPCA case, by exploiting again the idea of the `Carrier` and of a similar construction. Clearly, also optimization strategies should be extended to the vectorial case, in particular the finite differences method should be extended in order to find a good approximation of the Jacobian of the function, in order to avoid to lose the high order of convergence of the Exact Newton's method.

In conclusion, another possible extension of the code could be in managing the `DOF_matrix`. So far the user can give as input the `DOF_matrix` already computed, but to be accepted this matrix has to be *full*, in the sense that it has to contain all the Degrees Of Freedom for each λ present in the vector `lambda` given as input. In a future version of the code this could be made less restrictive and a partially filled `DOF_matrix` could be accepted, in order to avoid the Degrees Of Freedom computation for the already given entries, and completing the other missing values in a shorter amount of time.

Bibliography

[1] Ardenghi G., Vicini A.
Space-time regression models with differential regularization.
APSC report; November 2019.

[2] Azzimonti L., Sangalli L.M., Secchi P:, Domanin M., Nobile F.
Blood flow velocity field estimation via spatial regression with PDE penalization.
J. Amer. Statist. Assoc. **110** (511), 1057-1071; 2015.

[3] Bambini C., Giussani L.
Regression Models with Differential Regularization - Efficient computation of the Equivalent Degrees of Freedom.
APSC report; 2017.

[4] Colli A., Colombo L.
fdaPDE 1.0
APSC report; January, 2019.

[5] Colombo A., Perin G.
GSR-PDE.
APSC report; June 2020.

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

[7] Ghilotti L., Pigolotti C.
Density estimation with differential regularization.
APSC report; February 26th, 2020.

[8] Kim J.
fdaPDE: Tree Search Algorithm.
APSC report; March 2, 2020.

[9] Negri L.
Functional Principal Component Analysis Over Volumetric Domains with Neuroimaging Applications.
Thesis; 2018.

[10] Negri L.
Smooth Functional Principal Component Analysis-Regularization technique for data distributed over planar and two-dimensional manifold domains
APSC report; March 19, 2018.

[11] Quarteroni, A., Sacco, R., Saleri, F., Gervasio, P.
Matematica Numerica, 4^a Ed.
Springer, Berlin Heidelberg; 2013.

[12] R Core Team
Writing R extensions - Version 3.6.1
`https://cran.r-project.org/doc/manuals/r-release/R-exts.html`

[13] Ramsay T.
Spline smoothing over difficult regions.
J. R. Stat. Soc. Ser. B. Stat. Methodol. **64**(2), 307-319; 2002.

[14] Sangalli, L.M.
A novel approach to the analysis of spatial and functional data over complex domains.
Qual. Eng. **32**(2), 181-190; 2020.

[15] Sangalli L.M., Ramsay J.O., and Ramsay T.O.
Spatial spline regression models.
J. R. Stat. Soc. Ser. B. Stat. Methodol. **75**(4), 681-703; 2013.

[16] Shlesinger E.
Algebra lineare e geometria, 1^a Ed.
Zanichelli; 2011.

[17] Wood S. N., Bravington M. V., Hedley S. L.
Soap film smoothing.
J. R. Stat. Soc. Ser. B. Stat. Methodol., **70**(5), 931-955; 2008.

[18] How to install R on Ubuntu:
`https://www.digitalocean.com/community/tutorials/how-to-install-r-on-ubuntu-18-04`

[19] More details about `Rprintf`:
`https://teuder.github.io/rcpp4everyone_en/060_utility.html`

[20] More details about Roxygen:
`https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html`

[21] More details about Triangle:
Triangle: A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator.
`https://www.cs.cmu.edu/~quake/triangle.html`

Appendices

Appendix A

New `src` Structure

Code

- **C_Libraries**
  - FEMtriangulation.c
  - triangle.c
  - triangle.h
- **Density_Estimation**
  - **Include**
    - Data_Problem.h
    - Data_Problem_imp.h
    - DE_Data.h
    - Density_Initialization.h
    - Density_Initialization_Factory.h
    - Density_Initialization_imp.h
    - Descent_Direction.h
    - Descent_Direction_Factory.h
    - Descent_Direction_imp.h
    - FE_Density_Estimation.h
    - FE_Density_Estimation_imp.h
    - Functional_Problem.h
    - Functional_Problem_imp.h
    - K_Fold_CV_L2_Error.h
    - K_Fold_CV_L2_Error_imp.h
    - Optimization_Algorithm.h
    - Optimization_Algorithm_Factory.h
    - Optimization_Algorithm_imp.h
    - Preprocess_Factory.h
    - Preprocess_Phase.h
    - Preprocess_Phase_imp.h
  - **Source**
    - DE_Data.cpp
    - Rfun_Density_Estimation.cpp
- **FE_Assemblers_Solvers**
  - **Include**
    - Evaluator.h
    - Evaluator_imp.h
    - Finite_Element.h
    - Finite_Element_imp.h
    - Integrate_Psi.h
    - Integration.h
    - Kronecker_Product.h
    - Matrix_Assembler.h
    - Matrix_Assembler_imp.h
    - Param_Functors.h
    - Projection.h
    - Projection_imp.h
    - Solver.h
    - Spline.h
  - **Source**
    - FEM_Eval.cpp
    - Integration.cpp
    - Kronecker_Product.cpp

Code

- **FPCA**
  - **Include**
    - FPCA_Data.h
    - FPCA_Object.h
    - Mixed_FE_FPCA.h
    - Mixed_FE_FPCA_Factory.h
    - Mixed_FE_FPCA_imp.h
  - **Source**
    - FPCA_Data.cpp
    - FPCA_Object.cpp
    - Rfun_Smooth_FPCA.cpp
- **Global_Utilities**
  - **Include**
    - Factory.h
    - Make_Unique.h
    - Proxy.h
    - Solver_Definitions.h
    - Timing.h
- **Lambda_Optimization** [new features introduced]
  - **Include**
    - Auxiliary_Optimizer.h
    - Auxiliary_Optimizer_imp.h
    - Carrier.h
    - Function_Variadic.h
    - Gof_Updater.h
    - Grid_Evaluator.h
    - Lambda_Optimizer.h
    - Lambda_Optimizer_imp.h
    - Newton.h
    - Newton_imp.h
    - Optimization_Data.h
    - Optimization_Methods_Factory.h
    - Solution_Builders.h
    - Solution_Builders_imp.h
  - **Source**
    - Auxiliary_Optimizer.cpp
    - Optimization_Data.cpp

Code

- **Mesh**
  - **Include**
    - AD_Tree.h
    - AD_Tree_imp.h
    - Bounding_Box.h
    - Bounding_Box_imp.h
    - Domain.h
    - Domain_imp.h
    - Exception_Handling.h
    - Mesh.h
    - Mesh_imp.h
    - Mesh_Objects.h
    - Mesh_Objects_imp.h
    - Tree_Header.h
    - Tree_Header_imp.h
    - Tree_Node.h
    - Tree_Node_imp.h
  - **Source**
    - Examp.cpp
    - Mesh_Objects.cpp
- **Regression**
  - **Include**
    - FPIRLS.h
    - FPIRLS_Factory.h
    - FPIRLS_imp.h
    - Mixed_FE_Regression.h
    - Mixed_FE_Regression_imp.h
    - Regression_Data.h
    - Regression_Data_imp.h
  - **Source**
    - Regression_Data.cpp
    - Rfun_Auxiliary.cpp
    - Rfun_Regression_Laplace.cpp
    - Rfun_Regression_PDE.cpp
    - Rfun_Regression_PDE_Space_Varying.cpp
- **Skeletons**
  - **Include**
    - Auxiliary_Skeleton.h
    - DE_Initialization_Skeleton.h
    - DE_Skeleton.h
    - FPCA_Skeleton.h
    - GAM_Skeleton.h
    - Regression_Skeleton.h
    - Regression_Skeleton_Time.h
- FdaPDE.h
- FdaPDE_init.h
- install.libs.R
- Makevars

Diagram of the new sub-directories, `R` called files are highlighted in *orange*.

Appendix B

`preapply` – `apply` Functions

In this appendix we add the structure of the core functions of the regression problem `preapply` and `apply` for possible reference.

```
template<typename InputHandler>
template<UInt ORDER, UInt mydim, UInt ndim, typename IntegratorSpace, typename
    IntegratorTime, UInt SPLINE_DEGREE, UInt ORDER_DERIVATIVE, typename A>
void MixedFERegressionBase<InputHandler>::preapply(EOExpr<A> oper,
   const ForcingTerm & u, const MeshHandler<ORDER, mydim, ndim> & mesh_)
{
  const MatrixXr * Wp = regressionData_.getCovariates();

  UInt nnodes = N_*M_; // total number of spatio-temporal nodes
  FiniteElement<IntegratorSpace, ORDER, mydim, ndim> fe;

  // Set Areal data if present and not already done
  if(regressionData_.getNumberOfRegions()>0 && !isAComputed)
  {
     this->template setA<ORDER, mydim, ndim>(mesh_);
     isAComputed = true;
  }

  // Set psi matrix if not already done
  if(!isPsiComputed){
     this->template setPsi<ORDER, mydim, ndim>(mesh_);
     isPsiComputed = true;
  }

  // If there are covariates in the model set H and Q
  if(Wp->rows() != 0)
  {
     setH();
     setQ();
  }

  typedef EOExpr<Mass> ETMass; Mass EMass; ETMass mass(EMass);
  if(!isR1Computed)
  {
     Assembler::operKernel(oper, mesh_, fe, R1_);
     isR1Computed = true;
  }
  if(!isR0Computed)
  {
     Assembler::operKernel(mass, mesh_, fe, R0_);
     isR0Computed = true;
  }

```

```
  if(this->isSpaceVarying)
  {
     Assembler::forcingTerm(mesh_, fe, u, rhs_ft_correction_);
  }

  if(regressionData_.isSpaceTime())
  {
     this->template buildSpaceTimeMatrices<IntegratorTime, SPLINE_DEGREE,
         ORDER_DERIVATIVE>();
  }

  // Set final transpose of Psi matrix
  setpsi_t_();
  // Set matrix DMat for all cases
  setDMat();

  // Define right hand data [rhs]
  VectorXr rightHandData;
  getRightHandData(rightHandData); //updated
  this->_rightHandSide = VectorXr::Zero(2*nnodes);
  this->_rightHandSide.topRows(nnodes)=rightHandData;

  // Debugging purpose
  //Rprintf("Preliminary problem matrices building phase completed\n");
}
```

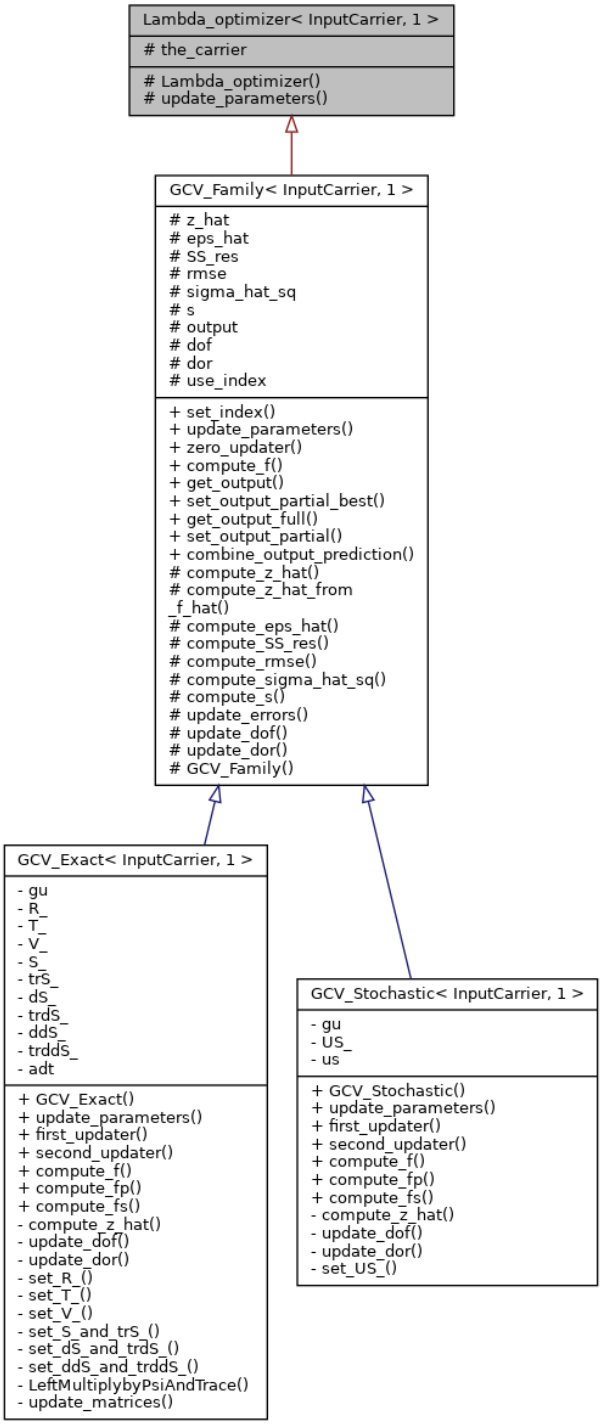
```
template<typename InputHandler>
MatrixXv MixedFERegressionBase<InputHandler>::apply(void)
{
  UInt nnodes = N_*M_; // Define nuber of nodes
  const VectorXr * obsp = regressionData_.getObservations(); // Get observations

  UInt sizeLambdaS;
  if (!regressionData_.isSpaceTime() && !isGAMData)
     sizeLambdaS=1;
  else
     sizeLambdaS = optimizationData_.get_size_S();
  UInt sizeLambdaT = optimizationData_.get_size_T();

  this->_solution.resize(sizeLambdaS,sizeLambdaT);
  this->_dof.resize(sizeLambdaS,sizeLambdaT);
  this->_GCV.resize(sizeLambdaS,sizeLambdaT);
  if(regressionData_.getCovariates()->rows() != 0)
  {
     this->_beta.resize(sizeLambdaS,sizeLambdaT);
  }

  VectorXr rhs = _rightHandSide; // Save rhs for modification

  for(UInt s=0; s<sizeLambdaS; ++s)
  {
     for(UInt t=0; t<sizeLambdaT; ++t)
     {
                     Real lambdaS;
        if(!regressionData_.isSpaceTime() && !isGAMData) //at the moment only space
            is implemented
           {
              lambdaS = optimizationData_.get_current_lambdaS();
           }
        else
```

```
34          lambdaS = (optimizationData_.get_lambda_S())[s];
35
36      Real lambdaT = (optimizationData_.get_lambda_T())[t];
37      _rightHandSide = rhs;
38
39      if(isGAMData || regressionData_.isSpaceTime() ||
            optimizationData_.get_current_lambdaS() !=
            ptimizationData_.get_last_lS_used())
40      {
41         if(!regressionData_.isSpaceTime())
42         {
43            buildSystemMatrix(lambdaS);
44         }
45                     else
46         {
47            buildSystemMatrix(lambdaS, lambdaT);
48         }
49      }
50
51      // Right-hand side correction for space varying PDEs
52      if(this->isSpaceVarying)
53      {
54          _rightHandSide.bottomRows(nnodes)= (-lambdaS)*rhs_ft_correction_;
55      }
56
57      // Right-hand side correction for initial condition in parabolic case
58      if(regressionData_.isSpaceTime() && regressionData_.getFlagParabolic())
59      {
60         for(UInt i=0; i<regressionData_.getInitialValues()->rows(); i++)
61         {
62            _rightHandSide(nnodes+i) -= lambdaS*rhs_ic_correction_(i);
63         }
64      }
65
66      // Applying boundary conditions if necessary
67      if(regressionData_.getDirichletIndices()->size() != 0) // if areal data NO
            BOUNDARY CONDITIONS
68         addDirichletBC();
69
70
71      // Factorization of the system for woodbury decomposition
72      if(isGAMData || regressionData_.isSpaceTime() ||
            optimizationData_.get_current_lambdaS() !=
            optimizationData_.get_last_lS_used())
73      {
74         system_factorize();
75      }
76
77      // system solution
78      _solution(s,t) = this->template system_solve(this->_rightHandSide);
79
80
81      if(optimizationData_.get_loss_function()=="GCV" &&
            (!isGAMData&&regressionData_.isSpaceTime()))
82      {
83         if (optimizationData_.get_DOF_evaluation()!="not_required")
84         {
85            computeDegreesOfFreedom(s,t,lambdaS,lambdaT);
86         }
87         computeGeneralizedCrossValidation(s,t,lambdaS,lambdaT);
88      }
```

```
89          else
90          {
91             _dof(s,t) = -1;
92             _GCV(s,t) = -1;
93          }
94
95          // Covariates computation
96          if(regressionData_.getCovariates()->rows()!=0)
97          {
98             MatrixXr W(*(this->regressionData_.getCovariates()));
99             VectorXr P(*(this->regressionData_.getWeightsMatrix()));
100            VectorXr beta_rhs;
101            if( P.size() != 0)
102            {
103               beta_rhs = W.transpose()*P.asDiagonal()*(*obsp -
                      psi_*_solution(s,t).topRows(psi_.cols()));
104            }
105            else
106            {
107               beta_rhs = W.transpose()*(*obsp -
                      psi_*_solution(s,t).topRows(psi_.cols()));
108            }
109            _beta(s,t) = WTW_.solve(beta_rhs);
110         }
111      }
112   }
113   if(!(isGAMData||regressionData_.isSpaceTime()) &&
          optimizationData_.get_current_lambdaS()!=optimizationData_.get_last_lS_used())
114   {
115      optimizationData_.set_last_lS_used(optimizationData_.get_current_lambdaS());
116   }
117   _rightHandSide = rhs; // Return rhs to original status for next apply call
118
119   return this->_solution;
120 }
```

Appendix C

`Lambda_Optimization` Class Hierarchy

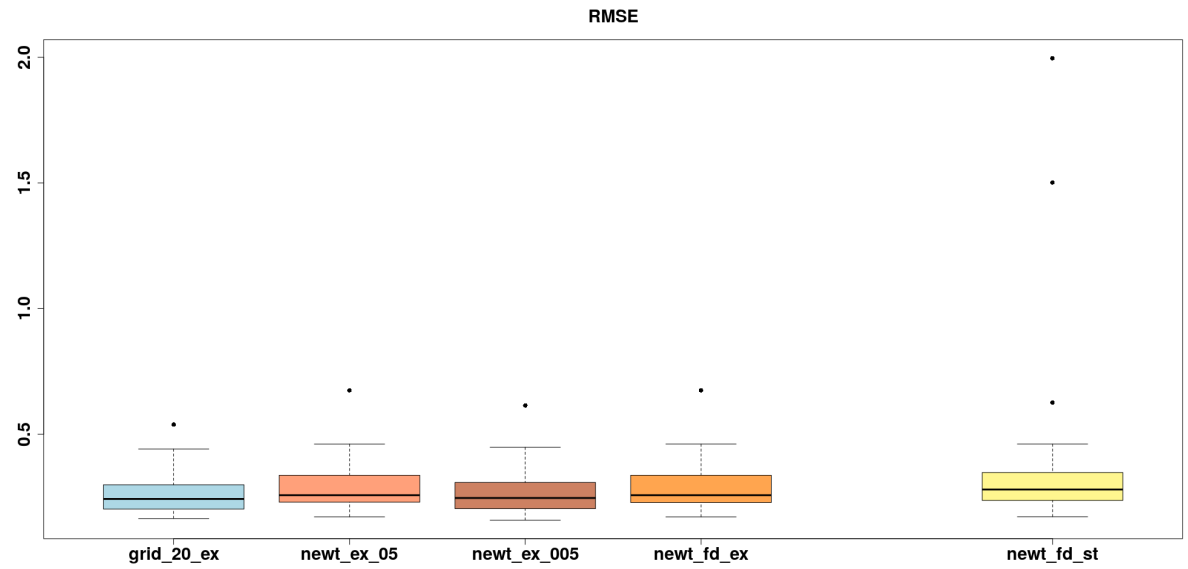


Appendix D

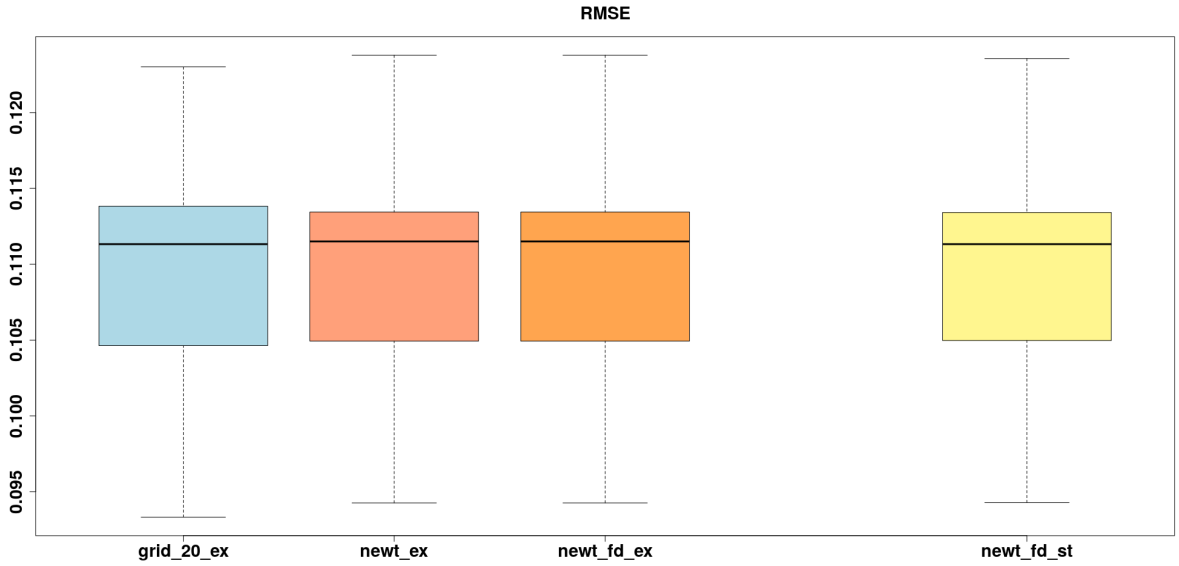
Appendix to the Simulation Studies

We report here some minor tests and plots of the testing Section 5 without the presence of those cases which are out of scale (e.g., the grid stochastic case is often out of scale for what concerns the RMSE values).

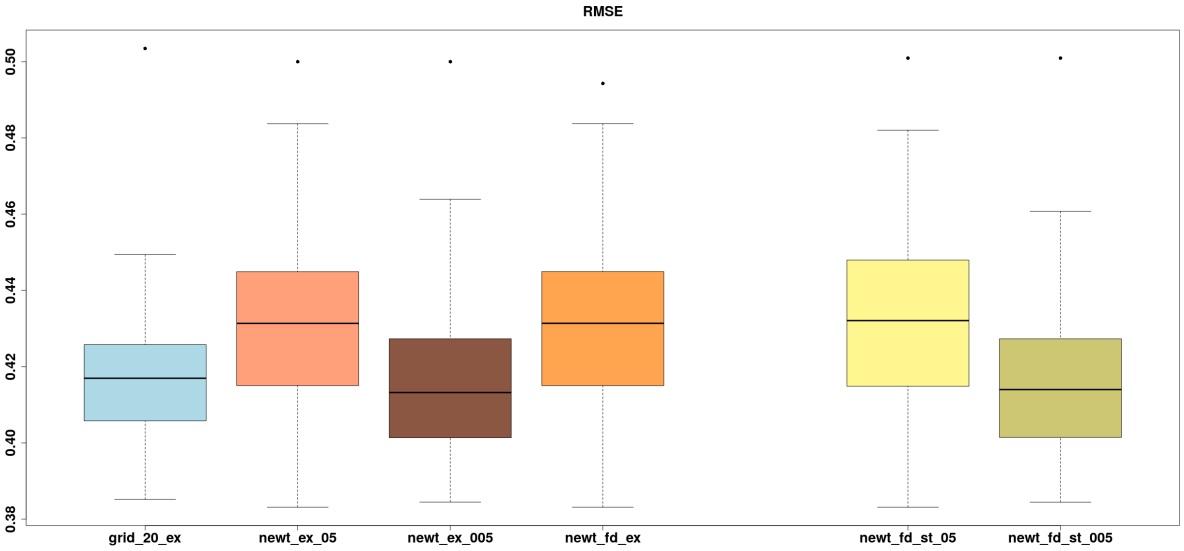
D.1 Additional plots



(a) Plot of RMSE from the test of Section 5.1.2 (Mesh C with covariates), without the grid with GCV stochastic case. If we look at the boxplot, we notice that the RMSE is similar in all the cases, confirming the stability of the Newton's finite differences method, with respect, e.g., to the stochastic grid evaluation.



(b) Plot of RMSE from the test of Section 5.2 (2.5D sphere), without the grid with GCV stochastic case.



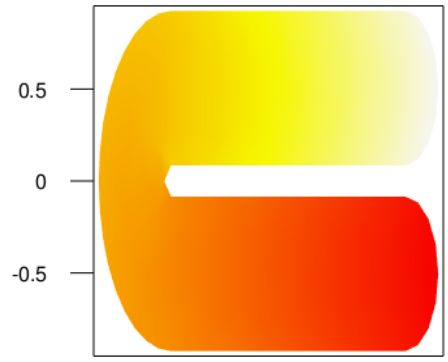
(c) Plot of RMSE from the test of Section 5.3, without the grid with GCV stochastic case. We observe that with a tolerance of 0.005 we obtain a much better RMSE, in both the Newton's exact method with exact GCV and Newton's finite differences one with stochastic GCV.

Figure D.1: Additional plots from testing Section.

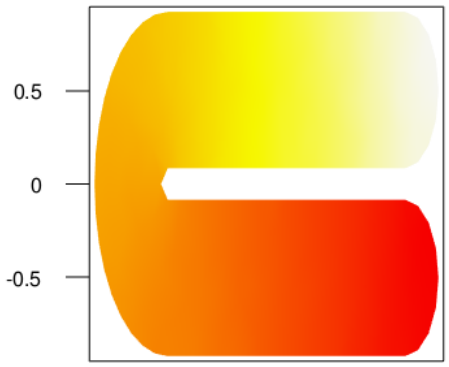
D.2 2D smoothing

D.2.1 Mesh C (horseshoe)

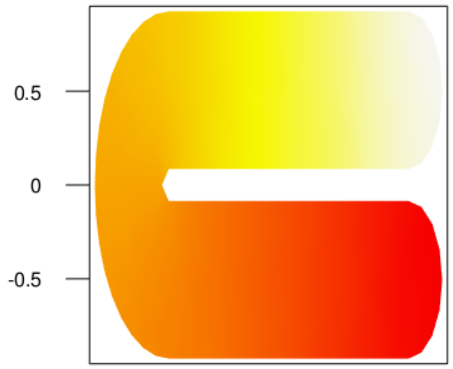
Mesh C - Test 6 [264 nodes - 171 observations - no covariates - point locations not at nodes]



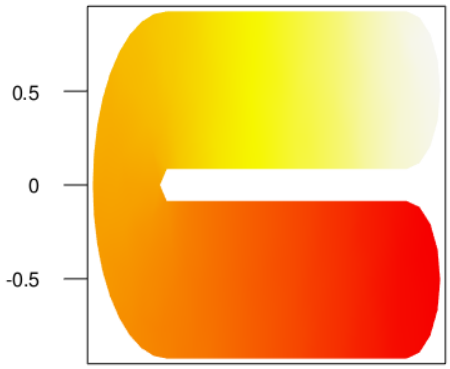
(a) true field



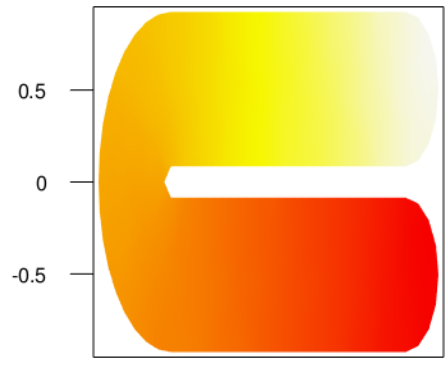
(b) `newt_ex`



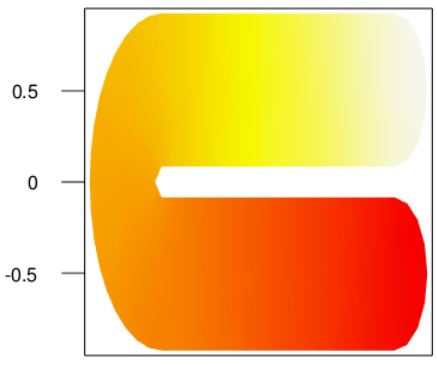
(c) `grid_20_ex`



(d) `grid_20_st`



(e) `newt_fd_st_0.05`

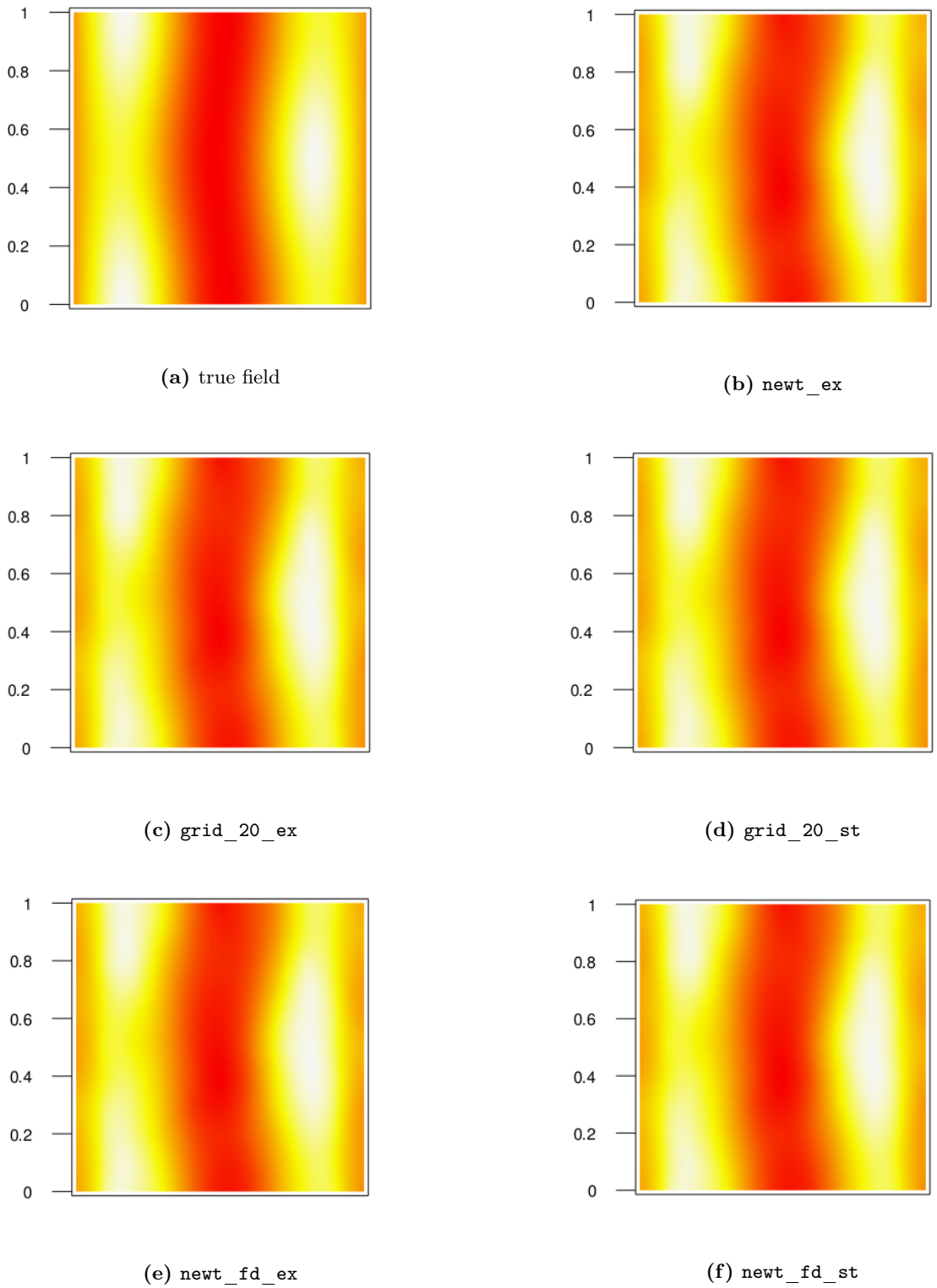


(f) `newt_fd_st_0.005`

Figure D.2: Test 6 (C mesh): comparison among the different output fields

D.2.2 Mesh PDE

Mesh PDE - Test 7 [441 nodes - 441 observations - point location at nodes]



(a) true field

(b) `newt_ex`

(c) `grid_20_ex`

(d) `grid_20_st`

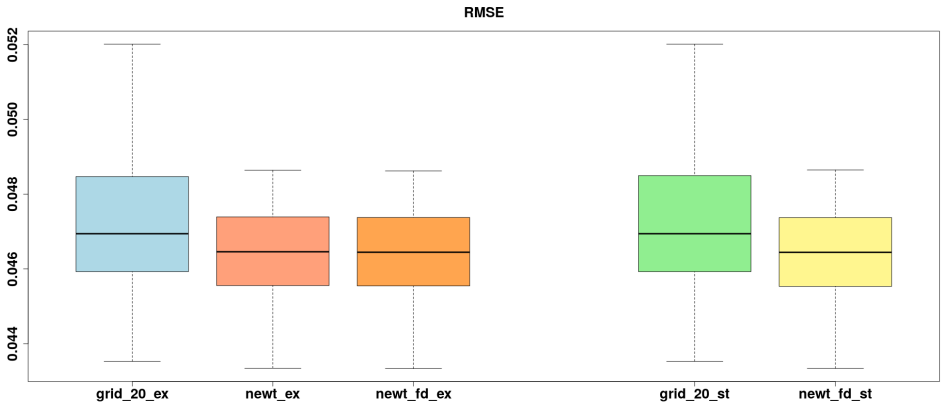
(e) `newt_fd_ex`

(f) `newt_fd_st`

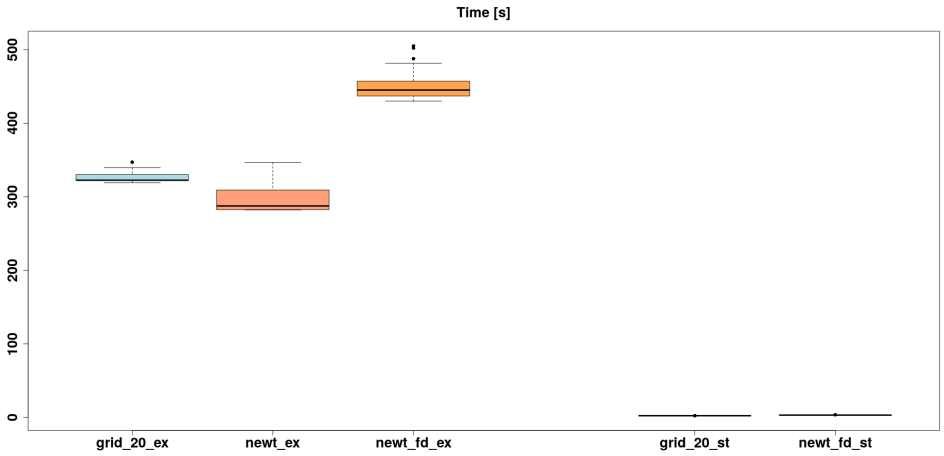
Figure D.3: Test 7 (mesh PDE): comparison among the different output fields

D.2.3 Square domain

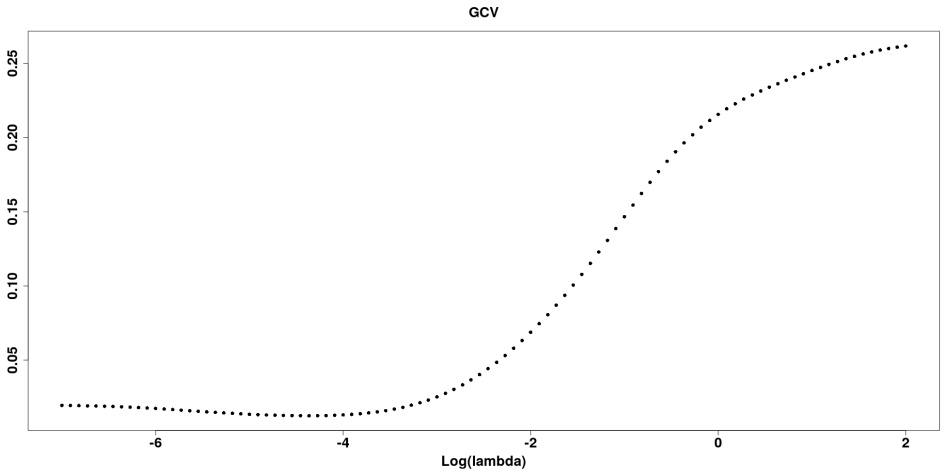
Square domain - Test 2 [3'281 nodes - 3'281 observations - point locations at nodes]



(a) Method vs RMSE



(b) Method vs Time

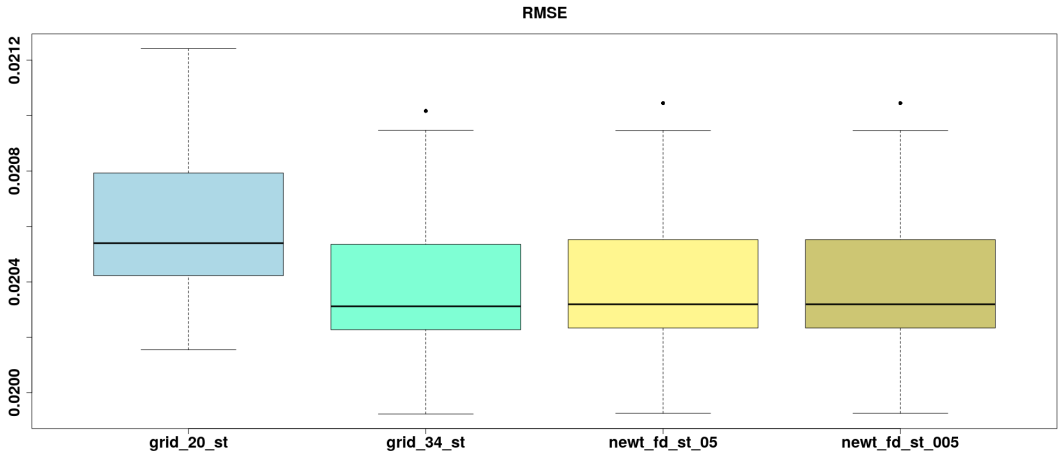


(c) GCV as function of λ

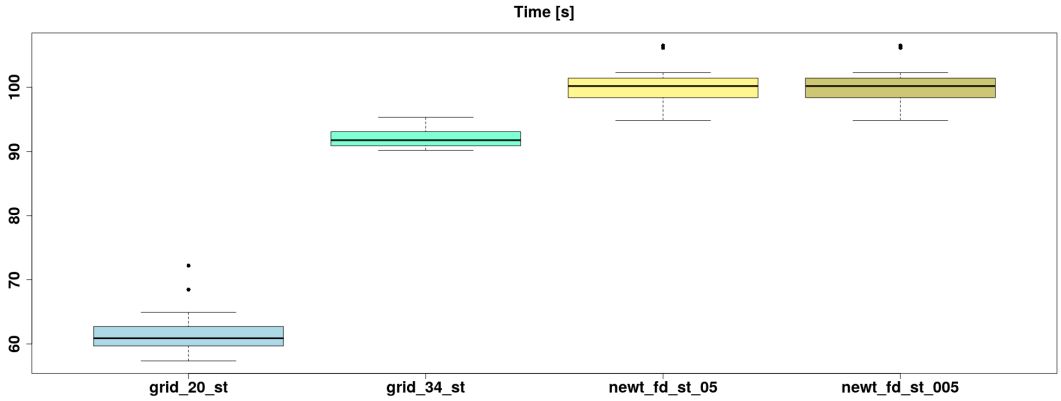
Figure D.4: Test 2 (square domain), 3'281 nodes - 3'281 observations - point locations at nodes

Method	Optimal λ	GCV	Average time
grid_20_ex	6.95e-05	1.269e-02	322.5
newt_ex	4.28e-05	1.254e-02	287.5
newt_fd_ex	4.28e-05	1.254e-02	445.1
grid_20_st	6.95e-05	1.267e-02	2.390
newt_fd_st	4.29e-05	1.250e-02	3.184

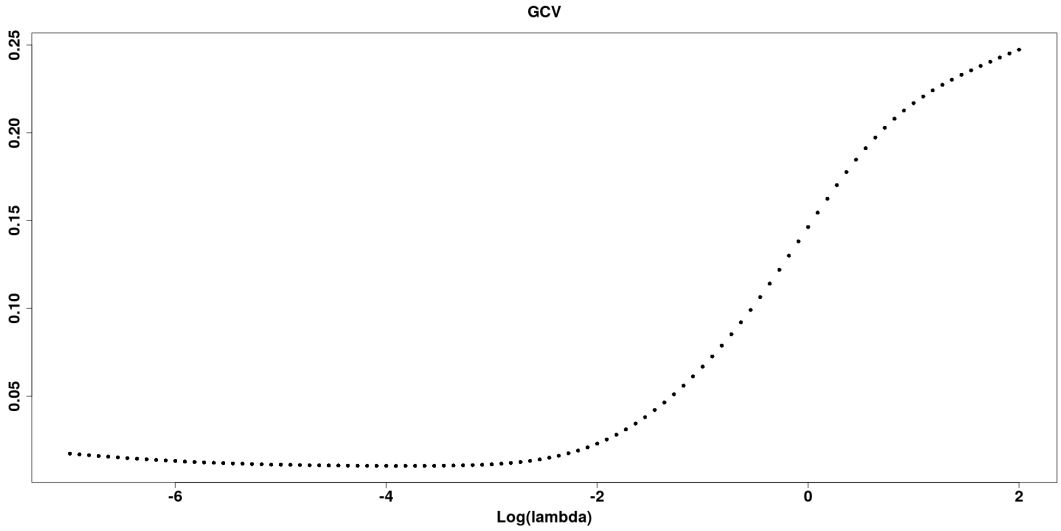
Square domain - Test 4 [30'971 nodes - 30'971 observations - point locations at nodes]



(a) Method vs RMSE



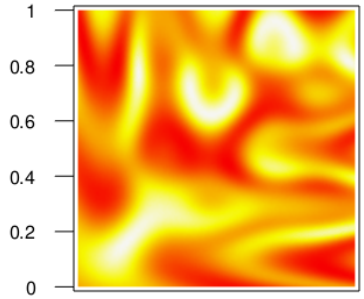
(b) Method vs Time



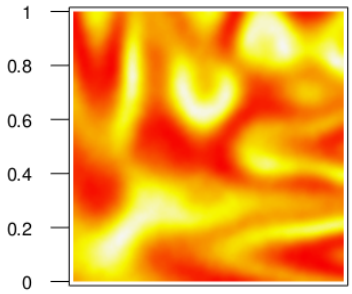
(c) GCV as function of λ

Figure D.5: Test 4 (square domain), 30'971 nodes - 30'971 observations - point locations at nodes

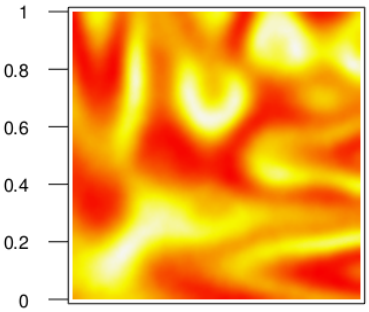
Method	Optimal λ	GCV	Average time
grid_20_st	2.1e-04	1.041e-02	60.88
grid_34_st	1.6e-04	1.040e-02	92.70
newt_fd_st_0.05	1.5e-04	1.040e-02	100.10
newt_fd_st_0.005	1.6e-04	1.040e-02	114.94



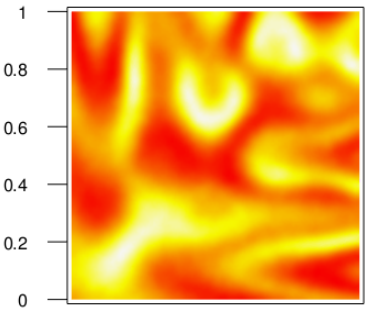
(a) true field



(b) `newt_fd_st_0.05`



(c) `grid_20_st`



(d) `grid_32_st`

Figure D.6: Test 4 (square domain): comparison among the different output fields