SQRPDE

Spatial Quantile regression with Partial Differential Equation Regularization

PACS Project
Academic Year 2022/2023

Authors: Marco Francesco De Sanctis, Ilenia Di Battista

POLITECNICO
MILANO 1863

Supervisors: Professor L.M. Sangalli, Professor E. Arnone

Contents

Chapter 1

Introduction

Regression is a statistical method broadly used in quantitative modeling. Researches typically use the values of several variables to explain or predict the mean values of a scalar outcome. However, in some scenarios the mean provides little information, especially when samples do not satisfy the Gaussian assumption. The goal of a quantile regression model is to estimate a generic conditional quantile of a univariate response, that is of particular interest when real data shows heteroscedasticity, skewness, fat or thin tails in their statistical distribution.

Spatial Regression with Partial Differential Equation regularization (SR-PDE) is an innovative statistical method which analyses data that display complicated spatial dependencies over a bounded domain, with regularizing terms involving partial differential equations.

Spatial Quantile Regression with Partial Differential Equation regularization (SQR-PDE), combining the main features of the two above mentioned models, faces the problem of estimating the conditional quantiles of data gathered on a spatial domain including a differential penalization term.

Conditional quantile functions are the object of interest in many fields, like risk assessment ([8]), high-dimensional learning ([12] and [11]) and in general in all those applications where there is an unequal variation of the dependent variable for different ranges of the predictors. Below we show an example of application of such kind of model to the Switzerland rainfall recorded in May 8, 1986. The results are taken from the phD thesis [3] by C.Castiglione.

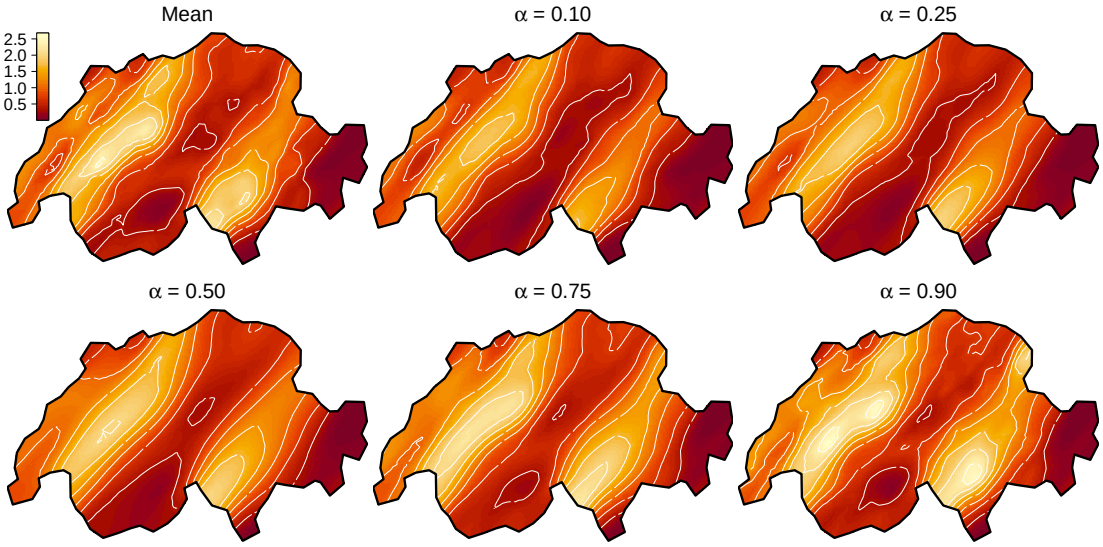


Figure 1.1: Estimated mean and α-quantile fields for the Switzerland rainfall data. Quantile surfaces differ from the mean one and across quantiles, with differences that are in localized regions. This means that some regions might expect more heterogenous patterns of rainfall than others, and more extreme events, even in the presence of similar mean rainfall [from C.Castiglione].

The main objective of this project is to implement the SQR-PDE model into the C++ library fdaPDE [7], starting from a previous R implementation ([4]). Differently from the R version of the code, which is based on a component-wise linear system resolution, in the new C++ version we will rely on the already existent Functional Penalized Iterative Reweighted Least Squares (FPIRLS) iterative algorithm, which allows us to find at each iteration the nonparametric and parametric part of the solution simultaneously. This, together with the particular emphasis that the C++ library gives to the efficiency, we have been able to achieve much better performances with respect to the previous R implementation.

1.1 Chapters' list

This work is organized in the following chapters:

Chapter 2 which aims at describing the mathematical framework for SQR-PDE and all its required theoretical tools.

Chapter 3 presents the code we have implemented in the new C++ library fdaPDE, describing its rationale and the way how we have organized it.

Chapter 4 shows the computational results concerning the SQR-PDE method and it is aimed at testing the goodness of the model and the improvements in terms of efficiency due to the new C++ implementation.

Chapter 5 sums up the obtained results, highlighting the strengths and the weaknesses of the proposed method.

Chapter 6 contains the instructions to install the fdaPDE library and to run the tests described in 4.

Chapter 2

Theoretical framework

2.1 Quantile regression

Quantile regression aims at expressing the conditional quantiles of a dependent variable Y as a linear function of the explanatory variables X, with the following simple mathematical model

$$Q_{Y|\mathbf{X}}(\alpha) = \mathbf{X}^T\boldsymbol{\beta}, \quad \alpha \in (0,1) \tag{2.1}$$

where α is the order of the quantile and $\boldsymbol{\beta}$ is the vector of the coefficients of the linear combination of the regressors. Differently from the classical regression, where the mean of the response is estimated, in quantile regression we can characterize the general conditional behaviour of the response variable by estimating any quantile of its distribution. This allows to treat samples which do not satisfy the Gaussian assumption required from the classical regression, having the additional benefit of being more robust to the presence of outliers.

2.2 SR-PDE

This section is meant to explain the modern SR-PDE class of statistical models. Who was familiar with this topic, can skip this section.

SR-PDE is a modern class of models to estimate the mean of spatially dependent data with a regularization term consisting of a Partial Differential Equation (PDE). The idea is to grasp the spatial correlation of the data embedding some prior knowledge of the complex phenomenon under study through the PDE, which permits to model spatio-temporal variations with strong anisotropies and non-stationarities, allowing for a great flexibility in the modelling phase. As described in [9], the other advantage of

this approach is that it can efficiently handle complex domains, with presence of holes or concavities and it can account for specific boundary conditions of the considered region. The mathematical model can be formulated as follows: let $\{y_i\}_i \subset \mathbb{R}$ be the values of the response variables in n spatial locations $\mathbf{p}_1, ..., \mathbf{p}_n$ scattered over a bounded spatial domain Ω with boundary of class $\mathcal{C}^2$. We consider the following semiparametric additive model

$$y_i = \mathbf{x}_i^T \boldsymbol{\beta} + f(\mathbf{p}_i) + \epsilon_i, \quad i = 1, ..., n \tag{2.2}$$

where $\boldsymbol{\beta} \in \mathbb{R}^q$ is the unknown vector of coefficients describing the effect of the covariates on the mean of the variable of interest, $f : \Omega \to \mathbb{R}$ is the unknown deterministic field describing the spatial relationship of the data and $\{\epsilon_1, ..., \epsilon_n\}$ are uncorrelated errors, with zero mean and finite variance. The solution is found minimizing the regularized sum-of-square-error

$$\sum_{i=1}^{n} (y_i - \mathbf{x}_i^T \boldsymbol{\beta} - f(\mathbf{p}_i))^2 + \lambda \int_{\Omega} (Lf - u)^2 \, d\mathbf{p} \tag{2.3}$$

where $\lambda > 0$ is the smoothing parameter controlling the amount of penalization and $(Lf - u)$ is the misfit of the field f with respect to the PDE.

L is a second order differential operator of the general form

$$Lf = div(\mathbf{K} \nabla f) + \mathbf{b} \cdot \nabla f + cf \tag{2.4}$$

where the tensor $\mathbf{K}$, the vector $\mathbf{b}$ and the scalar c model the diffusion, the advection and the reaction of the phenomenon under consideration, respectively. The forcing term u can be either null or a generic $L^2(\Omega)$ function for further flexibility of the model. The specification of the boundary conditions can be shortly written as

$$Bf = \gamma \ on \ \partial\Omega \tag{2.5}$$

where B is a differential operator, $\gamma : \partial\Omega \to \mathbb{R}$ is a smooth term on the boundary of the domain Ω.

The minimization problem 2.3 cannot be solved analitically, since it can be shown that the solution should be found as minimum of a fourth order problem. Therefore, one can proceed introducing the correspondent variational problem followed by its discretization and resolution via Finite Element Method (FEM).

2.3 SQR-PDE

SQR-PDE models are the extensions in the quantile regression framework of the SR-PDE ones. Let $\{Y_i\}_{i=1..n}$ be real independent random variables gathered on the spatial locations $\mathbf{p}_1, ..., \mathbf{p}_n$. Each Y_i has an absolutely continuous distribution with probability density function $\pi_{Y_i|\mathbf{p}_i}(y)$, cumulative density function $\Pi_{Y_i|\mathbf{p}_i}(y)$ and quantile function $Q_{Y_i|\mathbf{p}_i}(\alpha) = inf\{y \in \mathbb{R} : \Pi_{Y_i|\mathbf{p}_i}(y) \geq \alpha\}$ for any probability level $\alpha \in (0, 1)$.

We assume the following semiparametric spatial model for the $\alpha-$quantile of Y_i

$$Q_{Y_i|X_i,\mathbf{p}_i}(\alpha) = \mathbf{x}_i^T\boldsymbol{\beta} + f(\mathbf{p}_i), \quad i = 1, ..., n \tag{2.6}$$

where $\mathbf{x}_i \in \mathbb{R}^q$ is the vector of covariates observed at location $\mathbf{p}_i$, $\boldsymbol{\beta} \in \mathbb{R}^q$ is the vector of covariates effect and f is a spatial field over Ω as in the previous section. We estimate the unknown $\boldsymbol{\beta}$ and f by minimizing the penalized loss functional

$$J(\beta, f) = \frac{1}{n}\sum_{i=1}^{n} \rho_\alpha(y_i - \mathbf{x}_i^T\boldsymbol{\beta} - f(\mathbf{p}_i)) + \frac{\lambda}{2}\int_\Omega (Lf - u)^2 \, d\mathbf{p} \tag{2.7}$$

with the same meaning of the terms as in 2.2. Here, $\rho_\alpha(x) := \frac{1}{2}|x| + (\alpha - \frac{1}{2})x$ is the quantile check function, also called pinball loss. It is such that $Q_{Y_i|X_i,\mathbf{p}_i}(\alpha) = \underset{q\in\mathbb{R}}{\text{argmin}}\, \mathbb{E}[Y_i - q]$. The presence of the pinball loss in 2.7 makes the SQR-PDE problem much more difficult compared to SR-PDE. Indeed, the latter is based on a Least Squares problem, while the former is based on the minimization of a non-differentiable functional due to the presence of ρ_α.

It can be shown that the appropriate functional embedding space for the spatial field f in 2.7 is

$$\mathcal{F}_\gamma := \{f \in H^2(\Omega) : Bf = \gamma \; on \; \partial\Omega\} \tag{2.8}$$

Hence the estimation problem can be written as:

Problem 1 *Find* $(\hat{\boldsymbol{\beta}}, \hat{f}) \in (\mathbb{R}^q, \mathcal{F}_\gamma)$ *such that* $(\hat{\boldsymbol{\beta}}, \hat{f}) = \underset{\boldsymbol{\beta},f}{\text{argmin}}\; J(\boldsymbol{\beta}, f)$

The estimators $(\hat{\boldsymbol{\beta}}, \hat{f})$ must be computed via numerical methods and here we apply the Expectation-Maximization (EM) algorithm in order to approximate the optimization problem 1.

At the $(k + 1)-$th iteration of the algorithm, the estimates are updated following two

steps:

$$\begin{aligned} &\text{E-step:} \quad \text{Compute the expected penalized log-likelihood} \quad \boldsymbol{\ell}_p^{(k)}(\boldsymbol{\beta}, f; \mathbf{y}) \\ &\text{M-step:} \quad f^{(k+1)} = \underset{f \in \mathcal{F}_\gamma}{\operatorname{argmax}} \, \boldsymbol{\ell}_p^{(k)}(\boldsymbol{\beta}, f; \mathbf{y}) \end{aligned} \tag{2.9}$$

The expected penalized log-likelihood functional $\boldsymbol{\ell}_p^{(k)}(\boldsymbol{\beta}, f; \mathbf{y})$ can be written as $\boldsymbol{\ell}_p^{(k)}(\boldsymbol{\beta}, f; \mathbf{y}) = -nJ^{(k)}(\boldsymbol{\beta}, f) + const$, where $J^{(k)}(\boldsymbol{\beta}, f)$ is a local quadratic approximation of $J(\boldsymbol{\beta}, f)$ in a neighborhood of $(\boldsymbol{\beta}^{(k)}, f^{(k)})$ that is

$$J^{(k)}(\boldsymbol{\beta}, f) = \frac{1}{2n}(\mathbf{z}^{(k)} - \mathbf{X}\boldsymbol{\beta} - \mathbf{f}_n)^T \mathbf{W}^{(k)}(\mathbf{z}^{(k)} - \mathbf{X}\boldsymbol{\beta} - \mathbf{f}_n) + \frac{\lambda}{2}\int_\Omega (Lf - u)^2 \tag{2.10}$$

where $\mathbf{z}^{(k)} = \mathbf{y} - (1 - 2\alpha)|\mathbf{y} - \mathbf{X}\boldsymbol{\beta} - \mathbf{f}_n^{(k)}|$ is a vector of working observations, $\mathbf{W}^{(k)} = \text{diag}(\mathbf{w}^{(k)})$ is a working matrix and $1/\mathbf{w}^{(k)} = 2\ |\mathbf{y} - \mathbf{X}\boldsymbol{\beta} - \mathbf{f}_n^{(k)}|$.

Since the minimization of 2.10 is convex and quadratic, it can be solved extending the procedure discussed in 2.2. We can notice that the functional 2.10 is well-defined in the maximization step of the algorithm only if all the components of $1/\mathbf{w}^{(k)}$ are non-zero, namely $1/w_i^{(k)} \neq 0\ \forall i = 1, .., n$, otherwise the corresponding entries of $\mathbf{W}^{(k)}$ might diverge causing numerical instabilities. This is known as infinite-weight problem and it can be addressed enforcing the constraint $z_i^{(k)} - \mathbf{x}_i^T\boldsymbol{\beta} - f(\mathbf{p}_i) = 0$ for those i such that $w_i^k \to \infty$. Therefore, for a generic vector $\mathbf{a} \in \mathbb{R}^n$ we define the partition $\mathbf{a} = \{\mathbf{a}_s,\ \mathbf{a}_{-s}\}$ where $\mathbf{a}_s = \{a_i : 1/w_i = 0\}$ and $\mathbf{a}_s = \{a_i : 1/w_i \neq 0\}$. Hence the M-step of the EM algorithm can be alternatively stated as

Problem 2 *Find* $(\tilde{\boldsymbol{\beta}}, \tilde{f}) \in (\mathbb{R}^q, \mathcal{F}_\gamma)$ *such that* $(\tilde{\boldsymbol{\beta}}, \tilde{f}) = \underset{\boldsymbol{\beta}, f}{\operatorname{argmin}}\ J_{-s}^{(k)}(\boldsymbol{\beta}, f)$ *subject to* $(\mathbf{z}^{(k)} - \mathbf{X}\boldsymbol{\beta} - \tilde{\mathbf{f}}_n)_s = 0$

Thanks to the above formulation, the following constrained functional

$$J_{-s}^{(k)}(\boldsymbol{\beta}, f) = \frac{1}{2n}(\mathbf{z}^{(k)} - \mathbf{X}\boldsymbol{\beta} - \mathbf{f}_n)_{-s}^T \mathbf{W}_{-s}^{(k)}(\mathbf{z}^{(k)} - \mathbf{X}\boldsymbol{\beta} - \mathbf{f}_n)_{-s} + \frac{\lambda}{2}\int_\Omega (Lf - u)^2 \tag{2.11}$$

is always well-defined.

The formulation 2.11 is written in the form required by the FPIRLS algorithm, that is the generalization with PDE penalization of the PIRLS optimization algorithm, therefore we will exploit it to solve the minimization problem. For further details, we refer to [6]. Specifically, we focus on the unconstrained case, that is $\{i \in \{1...n\}\ s.t.\ 1/w_i = 0\} = \emptyset$. Therefore, the problem becomes

Problem 3 *At each iteration, find* $\underset{\boldsymbol{\beta}, \mathbf{f}}{\operatorname{argmin}}\ \left|\left|\frac{\mathbf{W}}{2n}^{\frac{1}{2}}(\mathbf{z} - \mathbf{X}\boldsymbol{\beta} - \mathbf{f}_n)\right|\right|^2 + \frac{\lambda}{2}\int_\Omega (Lf - u)^2$

Since Problem 3 does not admit analitical solution, we proceed with its discretization relying on the FEM procedure. To do so, first of all we need to write the variational formulation of such minimization problem. Let $g := Lf - u \in L^2(\Omega)$ be the field representing the misfit of the PDE and let $R_1(\cdot,\cdot)$, $R_0(\cdot,\cdot)$ and $F(\cdot)$ be defined as

$$\begin{aligned} R_1(\phi,\psi) &= \int_\Omega [(\mathbf{K}\nabla\phi)\cdot\nabla\psi + (\mathbf{b}\cdot\nabla\phi)\psi + c\phi\psi], \\ R_0(\phi,\psi) &= \int_\Omega \phi\psi, \\ F(\phi) &= \int_\Omega u\phi + \int_{\partial\Omega} \gamma\phi, \end{aligned} \tag{2.12}$$

for any pair of test functions $\phi, \psi \in H^1(\Omega)$.

The variational formulation of 3 is

Proposition 1 *Let $(\tilde{\boldsymbol{\beta}}, \tilde{f}) \in \mathbb{R}^q \times \mathcal{F}_\gamma$ be a minimum of $J^{(k)}$ and let $\tilde{g} = L\tilde{f} - u \in H^1(\Omega)$. Then $(\tilde{\boldsymbol{\beta}}, \tilde{f}, \tilde{g})$ must satisfy the following system*

$$\begin{aligned} -\mathbf{X}^T\mathbf{W}^{(k)}(\mathbf{z}^{(k)} - \mathbf{X}\tilde{\boldsymbol{\beta}} - \tilde{\mathbf{f}}_n) &= 0, \\ -(\boldsymbol{\psi}_n)^T\mathbf{W}^{(k)}(\mathbf{z}^{(k)} - \mathbf{X}\tilde{\boldsymbol{\beta}} - \tilde{\mathbf{f}}_n) + \lambda n R_1(\psi,\tilde{g}) &= 0, \\ R_1(\tilde{f},\phi) - R_0(\tilde{g},\phi) &= F(\phi), \end{aligned} \tag{2.13}$$

for any pair of test functions $\phi, \psi \in \mathcal{F}_0$.

Such variational formulation is the starting point for the derivation of the finite element discretization. We consider a triangularization $\Im_h$ of Ω, where h is the maximum length of the edges of the triangles. We call Ω_h the union of all the elements of $\Im_h$. The functions of the variational formulation can be approximated through functions in the finite element space $\mathcal{F}^r_{\gamma,h} := \{f_h \in \mathcal{C}^0(\overline{\Omega}_h) : f_{h|\tau} \in \mathcal{P}_r(\tau)\ \forall\tau \in \Im_h,\ Bf_h = \gamma_h \text{ on } \partial\Omega_h\} \subset H^1(\Omega_h) \cap \mathcal{C}^0(\overline{\Omega}_h)$. In particular we will consider the space $\mathcal{F}^1_{\gamma,h}$ and we indicate the elements of the basis expansion as $\psi_1, ..., \psi_N$ in correspondence of the mesh nodes $\boldsymbol{\xi}_1, ..., \boldsymbol{\xi}_N$.

Let $\boldsymbol{\Psi}$ be the $n \times N$ matrix evaluation of the N basis functions at the n data locations

$$\boldsymbol{\Psi} = \begin{bmatrix} \boldsymbol{\psi}^T(\mathbf{p}_1) \\ \vdots \\ \boldsymbol{\psi}^T(\mathbf{p}_n) \end{bmatrix} \tag{2.14}$$

Then for any $f_h \in \mathcal{F}^1_{\gamma,h}$, we have $f_h(\mathbf{p}) = \mathbf{f}^T\boldsymbol{\psi}(\mathbf{p})$ with $\boldsymbol{\psi}(\mathbf{p}) = (\psi_1(\mathbf{p}), ..., \psi_N(\mathbf{p}))^T$ and $\mathbf{f}_n = \boldsymbol{\Psi}\mathbf{f}$. Define by $\mathbf{R}_0$ and $\mathbf{R}_1$ the discretization of the bilinear forms $R_0(\cdot,\cdot)$ and $R_1(\cdot,\cdot)$ and by $\mathbf{u}$ and $\boldsymbol{\gamma}$ the discretization of $F(\cdot)$.

Then 2.13 can be approximated by the linear system

$$\begin{bmatrix} \mathbf{X}^T\mathbf{W}\mathbf{X} & \mathbf{X}^T\mathbf{W}\boldsymbol{\Psi} & \mathbf{0} \\ \boldsymbol{\Psi}^T\mathbf{W}\mathbf{X} & \boldsymbol{\Psi}^T\mathbf{W}\boldsymbol{\Psi} & \lambda n\mathbf{R}_1 \\ \mathbf{0} & \lambda n\mathbf{R}_1 & -\lambda n\mathbf{R}_0 \end{bmatrix} \begin{bmatrix} \tilde{\boldsymbol{\beta}} \\ \tilde{\mathbf{f}} \\ \tilde{\mathbf{g}} \end{bmatrix} = \begin{bmatrix} \mathbf{X}^T\mathbf{W}\mathbf{z} \\ \boldsymbol{\Psi}^T\mathbf{W}\mathbf{z} \\ \lambda n(\mathbf{u}+\boldsymbol{\gamma}) \end{bmatrix} \tag{2.15}$$

which is equivalent to

$$\begin{bmatrix} -\boldsymbol{\Psi}^T\mathbf{Q}\boldsymbol{\Psi} & \lambda\mathbf{R}_1^T \\ \lambda\mathbf{R}_1 & \lambda\mathbf{R}_0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{f}} \\ \hat{\mathbf{g}} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{\Psi}^T\mathbf{Q}\mathbf{y} \\ \lambda\mathbf{u} \end{bmatrix} \qquad \boldsymbol{\beta} = \left(\mathbf{X}^T\frac{\mathbf{W}}{2n}\mathbf{X}\right)^{-1}\mathbf{X}^T\frac{\mathbf{W}}{2n}(\mathbf{y}-\mathbf{f}_n) \tag{2.16}$$

where $\mathbf{H} = \mathbf{X}(\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}$ and $\mathbf{Q} = (\mathbf{I}-\mathbf{H})^T\frac{\mathbf{W}}{2n}(\mathbf{I}-\mathbf{H})$.

At this point we can exploit the symmetry of the matrix $\mathbf{W}$ and the symmetry and the idempotency of $(\mathbf{I}-\mathbf{H})$, to decompose the system in 2.16 as follows

$$\begin{bmatrix} -\boldsymbol{\Psi}^T\frac{\mathbf{W}}{2n}(\mathbf{I}-\mathbf{H})\boldsymbol{\Psi} & \lambda\mathbf{R}_1^T \\ \lambda\mathbf{R}_1 & \lambda\mathbf{R}_0 \end{bmatrix} = \begin{bmatrix} -\boldsymbol{\Psi}^T\frac{\mathbf{W}}{2n}\boldsymbol{\Psi} & \lambda\mathbf{R}_1^T \\ \lambda\mathbf{R}_1 & \lambda\mathbf{R}_0 \end{bmatrix} + \begin{bmatrix} -\boldsymbol{\Psi}^T\frac{\mathbf{W}}{2n}\mathbf{H}\boldsymbol{\Psi} & \mathbf{0}_N \\ \mathbf{0}_N & \mathbf{0}_N \end{bmatrix} =: \mathbf{E}+\mathbf{B} \tag{2.17}$$

where $\mathbf{0}_N$ is the matrix $\mathbb{R}^{N\times N}$ with all zero entries.

This is a very convenient form of the system since the matrix $\mathbf{E}$ does not include covariates, therefore it corresponds to the nonparametric part of the system, and it can be directly solved through LU factorization; instead the matrix $\mathbf{B}$ can be handled with Sherman Morrison Woodbury (SMW) formula writing $\mathbf{B} = \mathbf{U}_W\mathbf{C}_W\mathbf{V}_W$, where

$$\mathbf{U}_W := \begin{bmatrix} \boldsymbol{\Psi}^T\frac{\mathbf{W}}{2n} \\ \mathbf{0} \end{bmatrix} \quad \mathbf{C}_W := \left[\mathbf{X}^T\frac{\mathbf{W}}{2n}\mathbf{X}\right]^{-1} \quad \mathbf{V}_W := \left[\mathbf{X}^T\frac{\mathbf{W}}{2n}\boldsymbol{\Psi}\right] \tag{2.18}$$

Once we have identified those matrices we can apply the Woodbury identity stated in the following proposition.

Proposition 2

$$\mathbf{M}^{-1} := (\mathbf{E}+\mathbf{U}_W\mathbf{C}_W\mathbf{V}_W)^{-1} = \mathbf{E}^{-1} - \mathbf{E}^{-1}\mathbf{U}_W(\mathbf{C}_W^{-1}+\mathbf{V}_W\mathbf{E}^{-1}\mathbf{U}_W)^{-1}\mathbf{V}_W\mathbf{E}^{-1} \tag{2.19}$$

where $\mathbf{M}\in\mathbf{R}^{2N\times 2N}$, $\mathbf{E}\in\mathbf{R}^{2N\times 2N}$, $\mathbf{U}_W\in\mathbf{R}^{2N\times q}$, $\mathbf{C}_W\in\mathbf{R}^{q\times q}$ *and* $\mathbf{V}_W\in\mathbf{R}^{q\times 2N}$

The advantage in using SMW is that we have reduced the linear problem 2.16 to the computation of the inverse of $\mathbf{E}$ which is sparse, and thus not so computational expensive, and the inverse of $(\mathbf{C}_W^{-1}+\mathbf{V}_W\mathbf{E}^{-1}\mathbf{U}_W)$ which has dimension $q\times q$ which is

tipically small. In the case with no covariates, i.e. $q = 0$, we simply have $\mathbf{B} = \mathbf{E}$ hence we just have to solve a sparse system.

2.3.1 Smoothing parameter selection

The choice of the smoothing parameter λ in 2.7 is a critical issue, as it always happens in penalized regression models. Indeed, we have no clue on which could be the right value to be set, hence we should proceed more quantitatively with some statistical techniques. One of the most used approaches is the Generalized Cross-Validation (GCV) method, which consists in finding λ as the minimum of

$$GCV(\lambda) = \sum_{i=1}^{n} \frac{\rho_\alpha(y_i - \mathbf{x}_i^T\hat{\boldsymbol{\beta}} - \hat{f}_h(\mathbf{p}_i, \lambda))}{n - df(\lambda)} \tag{2.20}$$

where df are the so-called effective degrees of freedom. Its closed-form expression can also be written as

$$df = |s| + tr\{\mathbf{A}_c^{-1}(\mathbf{I} - \mathbf{C}_s^T\mathbf{B}_c^{-1}\mathbf{C}_s\mathbf{A}_c^{-1})(\mathbf{C}_{-s}^T\mathbf{W}_{-s}\mathbf{C}_{-s})\} \tag{2.21}$$

Notice that in case there are no active constraints to impose, the expression simplifies and it results in the usual expression of degrees of freedom for a weighted regression

$$df = tr\{\mathbf{A}_c^{-1}(\mathbf{C}_{-s}^T\mathbf{W}_{-s}\mathbf{C}_{-s})\} \tag{2.22}$$

It turns out that $df = tr(\mathbf{S})$, where $\mathbf{S}$ is the smoothing matrix defined as

$$\begin{aligned} \mathbf{S} &= \boldsymbol{\Psi}\mathbf{T}^{-1}\boldsymbol{\Psi}^T\mathbf{W} \\ \mathbf{T} &= \boldsymbol{\Psi}^T\mathbf{Q}\boldsymbol{\Psi} + \lambda\mathbf{R}_1^T\mathbf{R}_0^{-1}\mathbf{R}_1 \end{aligned} \tag{2.23}$$

2.3.2 Model extention: areal data

The SQR-PDE model described so far can be extended to deal with the areal data sampling case. In this case we no longer consider geostatistical data, meaning data observed at pointwise locations $\mathbf{p}_1, ..., \mathbf{p}_n$, but we start from data related to areal subdomains of Ω. We therefore consider $D_1, ..., D_n$ to be n disjoint subdomains of Ω. In each subdomain the model involve areal evaluations of the unknown field f which can be either mean or integral evaluations. From now on, when referring to the areal data setting, we will assume the following model for the $\alpha-$quantile of Y_i:

$$Q_{Y_i|X_i,\mathbf{p}_i}(\alpha) = \mathbf{x}_i^T\boldsymbol{\beta} + \frac{1}{|D_i|}\int_{D_i} f, \quad i = 1, ..., n \tag{2.24}$$

where $|D_i|$ denotes the area of D_i.

Once the model has been adapted we can focus on the numerical solution. In order to use FEM we need to redefine the matrix $\boldsymbol{\Psi}$ and to define the matrix $\mathbf{D}$ accounting for the measure of the subdomains as:

$$[\boldsymbol{\Psi}]_{ij} = \int_{D_i} \psi_j$$

$$\mathbf{D} = \begin{bmatrix} |D_1| & & \\ & \ddots & \\ & & |D_n| \end{bmatrix}$$

Applying such definitions the linear system 2.16 solved by FPIRLS becomes:

$$\begin{bmatrix} -\boldsymbol{\Psi}^T\mathbf{D}\mathbf{Q}\boldsymbol{\Psi} & \lambda\mathbf{R}_1^T \\ \lambda\mathbf{R}_1 & \lambda\mathbf{R}_0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{f}} \\ \hat{\mathbf{g}} \end{bmatrix} = \begin{bmatrix} -\boldsymbol{\Psi}^T\mathbf{D}\mathbf{Q}\mathbf{y} \\ \lambda\mathbf{u} \end{bmatrix} \tag{2.25}$$

Instead the $\boldsymbol{\beta}$ derivation stays the same.

Chapter 3

Code structure

The `fdaPDE` library is a C++ library meant to implement regression and functional statistical models with PDE regularization . The library is extremely modular, which means that one can focus on single modules during the development of the library in isolation with respect to the other modules avoiding the re-compilation the whole code-base. Indeed, the library is organized by layers in such a way that an implementation in a lower level layer is available for the upper ones. From the bottom to the upper part of the architecture we can distinguish 3 main blocks: core, calibration and models.

The structure of the core of `fdaPDE` originates by the problem itself, indeed any model based on the minimization of a functional has to deal with 3 problems: the discretization of the domain, the discretization of the differential operator L and the optimal choice of the smoothing parameter λ. Moreover, the discrete version of the variational problem leads to the solution of an appropriate linear system which has to be solved in the most efficient way. This reasoning defines the 4 modules of the core:

- `MESH`: provides an abstract geometrical representation of the domain to the external modules.
- `FEM`: manages the discretization of the bilinear form L and of the functional space where the solution is sought.
- `OPT`: collects different iterative schemes for the optimization of a field.
- `NLA`: includes numerical linear algebra algorithms (e.g. for the efficient resolution of linear systems).

Going up in the architecture we find the calibration module which, including the GCV and the K-FoldCV methods, handles the optimal selection of the hyperparameters of

the model. This module acts as a connection between the modules of the core and the statistical models. Indeed, on one hand it exploits the interface of the core for the optimization and on the other forces the models to expose a specific interface in order to make use of the calibration.

On top of the architecture there are the statistical models which can exploit all the modules below. The models are about 3 main areas: regression, functional and space-time. They are implemented as template classes and share some common methods.

We focus on the regression framework and the goal of our project is the implementation of the SQR-PDE model. Thus, we act at the model level of the library architecture as shown in figure 3.1.

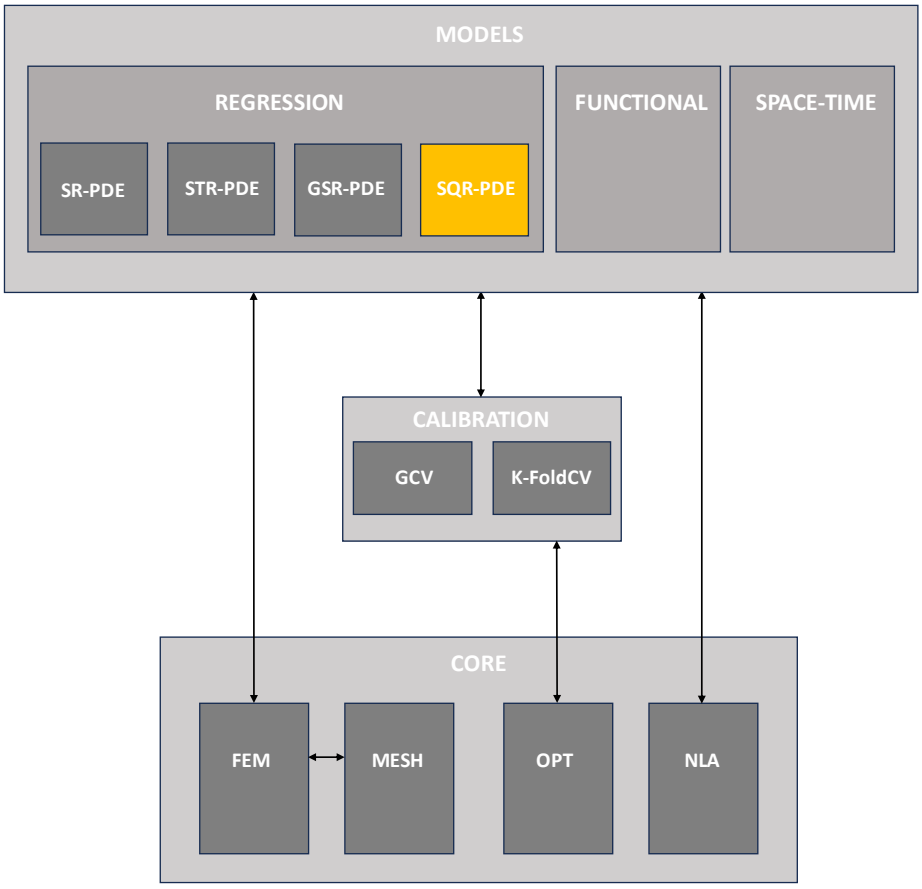


Figure 3.1: Architecture of the `fdaPDE` library with the insertion of our statistical model SQR-PDE (in orange).

The starting point of our work is a code for SQR-PDE models already developed in R by C.Castiglione. The main difference between the R version and our C++ code is the resolution of the linear problem 2.15. In R it is solved following an iterative component-wise approach meaning that, at each iteration, first $\boldsymbol{\beta}$ is computed using $\mathbf{f}$ evaluated in the previous step, then $\mathbf{f}$ is updated. Instead, in C++ the resolution of the system is handled through the FPIRLS algorithm and the SR-PDE solver, as it will be further detailed in the next section. In order to include SQR-PDE models within the library, the following tasks have to be fulfilled:

- Generalization of the FPIRLS algorithm
- Definition and declaration of the SQR-PDE class

Then two extensions of previous methods have been implemented:

- Alternative linear solver for the stochastic GCV computation
- Mass lumping implementation

Finally, we will briefly mention some changes operated on the Generalized Spatial Regression with Partial Differential Equation regularization (GSR-PDE) class to fix few issues and meet the modifications in the FPIRLS class.

3.1 Generalization of the FPIRLS algorithm

The FPIRLS class was designed to deal with the resolution of GSR-PDE problems, therefore some modification and extensions are necessary in order to manage also SQR-PDE problems and other possible future model classes which implement the `solve()` method relying on such algorithm.

The FPIRLS is a template class which takes as template arguments a model and a statistical distribution, required for the GSR-PDE case.

The first adjustment we made is to add the Gaussian to the set of the already existent distributions in `models::regression::Distributions.h` to deal with the SQR-PDE case.

```
1 class Gaussian {
2 private:
3   double mu_;
4   double sigma_;
5
6 public:
7   // constructor
```

```
    Gaussian() = default;
    Gaussian(double mu, double sigma) : mu_(mu), sigma_(sigma) {};
    // density function
    double pdf(double x) const { return 1/( std::sqrt(2*M_PI)*sigma_ ) *
        std::exp( -(x-mu_)*(x-mu_)/(2*sigma_*sigma_) ); };
    double mean() const { return mu_; }
    void preprocess(DVector<double>& data) const { return; }
    // vectorized operations
    DMatrix<double> variance(const DMatrix<double>& x) const { return
        DMatrix<double>::Ones(x.rows(), x.cols()); }
    DMatrix<double> link(const DMatrix<double>& x) const { return x; }
    DMatrix<double> inv_link(const DMatrix<double>& x) const { return x; }
    DMatrix<double> der_link(const DMatrix<double>& x) const { return
        DMatrix<double>::Ones(x.rows(), x.cols()); }

    // deviance function
    double deviance(double x, double y) { return (x-y)*(x-y); };

};
```

To avoid the definition of a useless private member `distribution_` in the model class, we have chosen to add the `Gaussian` parameter as default value for template parameter of the FPIRLS class.

```
template <typename Model, typename Distribution = Gaussian> class FPIRLS
```

As second generalization, we moved some calculations which previously were made inside the FPIRLS algorithm into each model that employs it, defining proper model functions. Doing so, FPIRLS just have to call those methods and each model can internally define its specific quantities. The cited methods and the associated calls inside FPIRLS are the following:

- `initialize_mu()`: initialization of the fitted part of the model for the iterative algorithm. It is saved in FPIRLS in the private member `mu_`:

```
    mu_ = m_.initialize_mu();
```

- `compute()`: calculation of the weight vector **pw**_ and the vector of pseudo-observations **py**_, extracted like:

```
    auto pair = m_.compute(mu_);
```

- `model_loss()`: computes the unpenalized part of the functional which is then assembled in FPIRLS as:

```
double J = m_.model_loss(mu_) + m_.lambdaS()*g_.dot(m_.R0()*g_);
```

The last and main modification of the FPIRLS class is the definition of the private member `solver_`. Indeed, in the previous implementation the solver object was defined inside the `compute()` method in `FPIRLS.h` and, as consequence, all the quantities of interest such as $\boldsymbol{\beta}, \mathbf{f}$ and $\mathbf{W}$ had first to be saved as private members of the FPIRLS class and then extracted through their getters within the `solve()` method of the model. This procedure generates unnecessary redondance, therefore the idea is to define the solver as private member of the FPIRLS class, initilize it inside the FPIRLS constructor and expose it through a getter. By doing so, the quantities of interest can be easily extracted through the getter of the solver itself and they no longer need to be saved within the FPIRLS class.

Thanks to these modifications, the FPIRLS algorithm can be now applied both in GSR-PDE and SQR-PDE problems and it is more general also in view of possible future new model classes.

3.2 SQR-PDE class

The SQR-PDE class is a template class which takes as template arguments a `PDE` object and a `SamplingDesign` strategy and which publicly inherites from the classes `RegressionBase` and `iGCV`. Such template structure is necessary to well-define the inheritance from `RegressionBase` which is itself a template class taking as parameters a model with that template structure.

The header file of the SQR-PDE class is reported below and follows the structure of other model classes like SR-PDE and GSR-PDE with some specific addition for the SQR-PDE problem.

```
namespace fdaPDE{
namespace models{

 template <typename PDE, typename SamplingDesign>
 class SQRPDE : public RegressionBase<SQRPDE<PDE, SamplingDesign>>,  ↩
     public iGCV {
   // compile time checks
   static_assert(std::is_base_of<PDEBase, PDE>::value);
 private:
   typedef RegressionBase<SQRPDE<PDE, SamplingDesign>> Base;
```

```
10
11    double alpha_;                                // quantile order
12    fdaPDE::SparseLU<SpMatrix<double>> invA_;     // factorization of A
13
14    DVector<double> py_{};  // pseudo observations
15    DVector<double> pW_{};  // diagonal of the weight matrix
16
17    // FPIRLS parameters (set to default)
18    std::size_t max_iter_ = 200;     // max number of iterations
19    double tol_weights_ = 1e-6;      // tolerance for the weights calculation
20    double tol_ = 1e-6;              // tolerance of the FPIRLS convergence
21
22    // utilities
23    double rho_alpha(const double&) const; // pinball loss function
24
25  public:
26    IMPORT_REGRESSION_SYMBOLS;
27    using Base::lambdaS; // smoothing parameter in space
28
29    // constructors
30    SQRPDE() = default;
31    SQRPDE(const PDE& pde, double alpha = 0.5) : Base(pde), alpha_(alpha) {};
32
33    // required by FPIRLS
34    DVector<double> initialize_mu() const;
35    std::tuple<DVector<double>&, DVector<double>&> compute(const
          DVector<double>& mu);
36    double model_loss(const DVector<double>& mu) const;
37
38    // setters
39    void setFPIRLSTolerance(double tol) { tol_ = tol; }
40    void setFPIRLSMaxIterations(std::size_t max_iter) { max_iter_ =
          max_iter; }
41
42    // ModelBase implementation
43    void init_model() { return; }
44    virtual void solve(); // finds a solution to the smoothing problem
45
46    // iGCV interface implementation
47    virtual const DMatrix<double>& T();
48    virtual const DMatrix<double>& Q();
49    double norm(const DMatrix<double>& obs, const DMatrix<double>& fitted)
          const;
50
```

```
51    // getters
52    const fdaPDE::SparseLU<SpMatrix<double>>& invA() const { return invA_; }
53    const DMatrix<double>& U() const { return U_; }
54    const DMatrix<double>& V() const { return V_; }
55    const bool massLumpingGCV() const { return Base::massLumpingGCV(); }
56
57    // Destructor
58    virtual ~SQRPDE() = default;
59  };
```

Given an overview of the class structure, now we briefly describe those methods which are specific for our class:

- `initialize_mu()`: computes the initializazion of the fitted part of the model

$$\boldsymbol{\mu} = \mathbf{f}_n + \mathbf{X}\boldsymbol{\beta}$$

  for the FPIRLS iterative algorithm.

- `compute()`: computes the weight and pseudo-observations vectors as follows:

$$\mathbf{pw}_{\_} = \frac{1}{2n(|\mathbf{y} - \boldsymbol{\mu}| + \texttt{tol\_weights\_})}$$

$$\mathbf{py}_{\_} = \mathbf{y} - (1 - 2\alpha)|\mathbf{y} - \boldsymbol{\mu}|$$

  where we highlight the usage of `tol_weights_` in the weight computation. This tolerance is added to the residual to prevent the denominator to be too small and cause numerical instabilities. This is a computational trick meant to meet the unconstrained formulation, as assumed in 2.3.

- `model_loss()`: computes the unpenalized loss of the model as

$$\texttt{J\_unpenalized} = ||W^{\frac{1}{2}}(\mathbf{py}_{\_} - \boldsymbol{\mu})||^2$$

- `T()`: required to support GCV-based smoothing parameter selection. It returns $\mathbf{T} = \boldsymbol{\Psi}^T\mathbf{Q}\boldsymbol{\Psi} + \lambda(\mathbf{R}_1^T\mathbf{R}_0^{-1}\mathbf{R}_1)$, eventually implemented with mass lumping as explained in the next section.

- `norm()`: compute the numerator of the GCV score as:

$$\texttt{norm}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}\texttt{rho\_alpha}(y_i - \hat{y}_i)$$

notice that a multiplicative constant is missing since it is added in the assembly of the GCV score expression in the method `eval()` of `GCV.h`.

It is worth to notice that in the class we do not have a private member which stores the weight matrix of the problem $\mathbf{W}$ and neither its getter; this is because the class is calling the member `W_` and the relative getter `W()` inherited by `RegressionBase`.

Finally we want to discuss the `solve()` method which is the one which actually finds the solution to the SQR-PDE problem. Its code is reported below together with the `compute` method of the FPIRLS class.

```
template <typename PDE, typename SamplingDesign>
void SQRPDE<PDE, SamplingDesign>::solve() {

  FPIRLS<decltype(*this)> fpirls(*this, tol_, max_iter_); // FPIRLS engine
  fpirls.compute();
  // fpirls converged: extract matrix W and solution estimates
  W_ = fpirls.solver().W();
  invA_ = fpirls.solver().invA();

  if(hasCovariates()){
    XtWX_ = X().transpose()*W_*X();
    invXtWX_ = XtWX_.partialPivLu();
    U_ = fpirls.solver().U();
    V_ = fpirls.solver().V();
    beta_ = fpirls.solver().beta();
  }

  f_ = fpirls.solver().f();

  return;
}
```

```
void compute() {

    static_assert(is_regression_model<Model>::value);

    mu_ = m_.initialize_mu();
    distribution_.preprocess(mu_);

    double J_old = tolerance_+1; double J_new = 0;

    // start loop
```

```
    while(k_ < max_iter_ && std::abs(J_new - J_old) > tolerance_){

    auto pair = m_.compute(mu_);

    // solve weighted least square problem
    solver_.data().template insert<double>(OBSERVATIONS_BLK,  ↩
        std::get<1>(pair));
    solver_.data().template insert<double>(WEIGHTS_BLK, std::get<0>(pair));

    // update solver to change in the weight matrix
    solver_.init_data();
    solver_.init_model();
    solver_.solve();

    // extract estimates from solver
    g_ = solver_.g();

    // update value of \mu_
    DVector<double> fitted = solver_.fitted();
    mu_ = distribution_.inv_link(fitted);

    // compute value of functional J for this pair (\beta, f)
    double J = m_.model_loss(mu_) + m_.lambdaS()*g_.dot(m_.R0()*g_);

    // prepare for next iteration
    k_++; J_old = J_new; J_new = J;

    }

    if (k_ == max_iter_)
    std::cout << "MAX ITER RAGGIUNTO " << std::endl;

    return;
}
```

As it is shown by the code, the SQR-PDE system in 2.15 is solved using the FPIRLS algorithm which looks for the minimum of the functional **J**. The `solve()` method first defines a FPIRLS engine, with the set tolerance and maximum number of iterations, and then relies on the `compute()` method of the FPIRLS class which actually implements its iterative scheme, as presented in the code. Specifically, it first initializes the vector $\boldsymbol{\mu}$ through the method `initialize_mu()` of the model, being this a model specific calculation, and then loops calling the `solve()` method on the SR-PDE solver until

convergence. The employment of the SR-PDE solver is enabled by the proper definitions of weights and pseudo-observations which allow to rewrite the SQR-PDE system in terms of a SR-PDE one. Indeed, the SR-PDE solver is initialized setting as data the weight vector **pw**_ and the vector of pseudo-observations **py**_ calculated by the method `compute()` of the SQR-PDE class. Thus, the `solve()` method of the SR-PDE class is called which solves the input system directly if the problem is nonparametric and via SMW if it is semiparametric. The FPIRLS convergence is reached when the variation of two successive values of the functional **J** is less than the set tolerance. If such condition does not happen, the algorithm stops when the maximum number of iterations is achieved. Once the FPIRLS algorithm has finished, the quantities of interest are extracted in the SQR-PDE `solve()` method through the getter of the FPIRLS member **solver**_, which is now exposed as explained in the previous section. In conclusion, at the end of the `solve()` method, we have the solution of the problem $(\boldsymbol{\beta}, f)$ and some quantities updated at convergence like the weight matrix $\mathbf{W}$, the inverse of the nonparametric matrix **invA** and, in case of a semiparametric problem, the Woodbury matrices $\mathbf{U}_W$ and $\mathbf{V}_W$ which are used in some of the methods for the GCV calculation.

3.3 Extentions

3.3.1 Alternative linear solver for the stochastic GCV computation

The computation of the GCV scores in 2.20 can be performed either exactly or via Monte Carlo approximation. The latter case is based on the following result

Proposition 3 *If* $\mathbf{U}$ *is a random vector s.t.* $\mathbb{E}[UU^T] = I$, *then* $\mathbb{E}[UAU^T] = tr(A)$

With the Monte Carlo method, we can approximate the above expected value with a finite sum over a chosen number of simulations, obtaining

$$tr(\mathbf{S}) = \mathbb{E}[\mathbf{U}\mathbf{S}\mathbf{U}^T] \approx \frac{1}{r}\sum_{i=1}^{r}(\mathbf{U}^{(i)})^T S\mathbf{U}^{(i)} \tag{3.1}$$

where $\mathbf{S}$ is the smoothing matrix, $\mathbf{U} \sim$ *Rademacher* and r is the chosen number of Monte Carlo simulations. The density function of $\mathbf{U}$ is

$$f(u) = \begin{cases} \frac{1}{2}, & u = 1 \\ \frac{1}{2}, & u = -1 \end{cases}$$

which means that $\mathbf{U} \sim$ *Rademacher* satisfies proposition 3. Recalling that $\mathbf{S}$ can be written as described in 2.23, we can rewrite 3.1 as $\sum_{i=1}^{r}(\mathbf{U}^T\boldsymbol{\Psi})^T(\mathbf{A}+\mathbf{U}_W\mathbf{C}_W\mathbf{V}_W)^{-1}\mathbf{B}$,

where $\mathbf{B} := -\mathbf{\Psi Q U}$, $\mathbf{U}_W, \mathbf{C}_W, \mathbf{V}_W$ are the Woodbury matrices described in 2.18. This formulation was already implemented in the `fdaPDE` library and it exploits the Woodbury decomposition taking advantage of the sparsity of the matrix $\mathbf{A}$. Alternatively, we can write 3.1 also as $\sum_{i=1}^{r}(\mathbf{U}^T\mathbf{\Psi})^T(\mathbf{\Psi}^T\mathbf{Q}\mathbf{\Psi} + \lambda\mathbf{R}_1^T\mathbf{R}_0^{-1}\mathbf{R}_1)^{-1}(-\mathbf{B})$ trying to leverage on the fact that the matrix to be inverted is positive definite, hence the Cholesky factorization can be used. We expect the SMW decomposition to perform better for large systems since, as described in [1], this technique avoids the inefficient inversion of the partially dense matrix 2.16; on the contrary, we expect the Cholesky implementation performing well on relatively small systems since in this case all the involved matrices have limited size. Indeed, in this scenario, the computational heavy step is the multiplication by the dense `nxn` matrix $\mathbf{Q}$, which is made efficient with the method `lmbQ()` employed by the Cholesky implementation. In this reguard, we have implemented the Cholesky decomposition in the class `StochasticEDF` as reported in the code below. We highlight that the factorization used is the lower triangular one, which is more efficient since the matrices are stored column major.

```
DMatrix<double> sol;   // room for problem solution
if(!model_.hasCovariates()){ // nonparametric case
sol = model_.invA().solve(Bs_);
} else{
    if (model_.n_basis() > N_threshold){
      // solve system (A+UCV)*x = Bs via woodbury decomposition using matrices U and V cached by model_
      sol = SMW<>().solve(model_.invA(), model_.U(), model_.XtWX(), model_.V(), Bs_);
    }
    else{
      // solve system (A+UCV)*x = Bs via Cholesky factorization
      Eigen::LLT<DMatrix<double>> lltOfA; // compute the Cholesky decomposition of A
      lltOfA.compute( model_.T() );
      sol = lltOfA.solve(- Bs_.topRows(n));
    }
}
```

As it is shown by the code, the compiler autonomously chooses which method to apply depending on the problem size. This implementation is justified by the results obtained in the tests described in section 4.2.1, where we present the results of the tests. The idea is that, analyzing the trade off between the memory consumption and the computational time, the Cholesky factorization performs better for relative small systems, instead the Woodbury decomposition is preferred for larger ones. Therefore, we define a constant

private member `N_threshold` inside the class `StochasticEDF` and, if `N < N_threshold`, the Cholesky algorithm is applied, otherwise the Woodbury one is performed.

Since the Cholesky algorithm brings some advantages in the stochastic approximation of the trace of $\mathbf{S}$, we decided to investigate its employment also in the resolution of the linear system 2.15 within the `solve()` method. However, as expected, the SMW algorithm has proven to be much more efficient, decreasing the time and the memory consumption needed to solve the system. This is coherent with the study carried out in [1], where the SMW algorithm is compared to other solvers like LU to solve SR-PDE problems. Indeed, our analysis explored the same framework of the cited paper since the resolution of SQR-PDE problems is based on an iterative resolution of a SR-PDE system and the Cholesky algorithm is comparable with the LU one in terms of efficiency. In conclusion, we discarded our Cholesky implementation for the `solve()` method and we relying only on the SMW one.

One of the reasons why the Cholesky factorization seems to work better than Woodbury in some scenarios of the GCV computation, but not at all at the level of the linear resolution in the `solve()` method, can be found analyzing more in details the structure of the `NLA::SMW.h` class. In particular, the computation of the vector $\mathbf{t}$ as $\mathbf{G}^{-1}\mathbf{V}\mathbf{y}$ is extremely slower when $\mathbf{b}$, and consequently $\mathbf{y}$, is passed to the SMW solver as a matrix instead as a vector. Since for the stochastic trace approximation we need to run M Monte Carlo simulations, when we call the SMW solver we pass a matrix $\mathbf{b}$ with M columns, and this causes the slow down discussed above. Below we show the already existent SMW class for the sake of clarity.

```
template <typename SparseSolver = fdaPDE::SparseLU<SpMatrix<double>>,
    typename DenseSolver = Eigen::PartialPivLU<DMatrix<double>>>
struct SMW{
  // constructor
  SMW() = default;

  // solves linear system (A + U*C^{-1}*V)x = b, assume to supply the already computed inversion of the dense matrix C.
  // This method must be executed after call to .compute()
  DMatrix<double> solve(const SparseSolver& invA, const DMatrix<double>& U, const DMatrix<double>& invC,
      const DMatrix<double>& V, const DMatrix<double>& b){
    DMatrix<double> y = invA.solve(b); // y = A^{-1}b
    // Y = A^{-1}U. Heavy step of the method. SMW is more and more efficient as q gets smaller and smaller
    DMatrix<double> Y = invA.solve(U);
    // compute dense matrix G = C^{-1} + V*A^{-1}*U = C^{-1} + V*y
```

```
    DMatrix<double> G = invC + V*Y;
    DenseSolver invG; invG.compute(G); // factorize qxq dense matrix G
    DMatrix<double> t = invG.solve(V*y);
    // v = A^{-1}*U*t = A^{-1}*U*(C^{-1} + V*A^{-1}*U)^{-1}*V*A^{-1}*b by solving linear system A*v = U*t
    DMatrix<double> v = invA.solve(U*t);
    return y - v; // return system solution
  }
};
```

3.3.2 Mass lumping implementation

The solution of system 2.16 requires the inversion of the mass matrix $\mathbf{R}_0$. This is a computationally heavy step and it becomes more and more demanding as the dimension of the system increases, since $\mathbf{R_0}$ is a dense matrix of dimension $N \times N$. The mass lumping technique is a strategy to reduce the number of calculation to invert a mass matrix: the idea is to define a diagonal or "lumped" approximation of the matrix itself as follows

$$\mathbf{R}_{0,ii}^{L} = \sum_{j=1}^{n} \mathbf{R}_{0,ij} \tag{3.2}$$

where $\mathbf{R}_0^L$ refers to the lumped version of $\mathbf{R}_0$. Obviously, once the mass matrix has been lumped, its inversion is much less demanding both in terms of computational time and memory effort.

In the code developed by C. Castiglione in R, since the computational effort due to the inversion of the matrix $\mathbf{R_0}$ is evident in terms of time, there is a fitting option `lumping` which, if set to `true`, treats the system exploiting the mass lumping technique. When such option is activated the observed speed up is significant in many different tests, therefore in R the default setting involve `lumping = True`. Following this lead, we implement such option also in the C++ code both to compare the R and C++ numerical solutions and to analyze if there is a reduction in the computational time also in C++ without a heavy worsening in the accuracy of the final result.

In order to do so, we act at the level of the assembly of the matrix $\mathbf{R_0}$. In particular we add a boolean `massLumpingSystem` in the `PDE.h` class which is set to `True`, through a proper setter, if we want to apply such strategy . The getter of such boolean is called in the initialization of the internal FEM solver, within the the method `init()` of `FEMBaseSolver.h`, so that the matrix $\mathbf{R_0}$ is computed accordingly.

```
  // initializes internal FEM solver status
  template <unsigned int M, unsigned int N, unsigned int R, typename E,
```

```
      typename F, typename B, typename I, typename S>
  void init(const PDE<M,N,R,E,F,B,I,S>& pde) {

    ...

    // compute mass matrix [R0]_{ij} = \int_{\Omega} \phi_i \phi_j by discretization of the identity operator
    R0_ = assembler.assemble(Identity());
    if(pde.massLumpingSystem()){
      DVector<double> R0_vector;
      R0_vector.resize(R0_.cols());
      for(std::size_t j = 0; j < R0_.cols(); ++j)
        R0_vector[j] = R0_.col(j).sum();
      R0_ = R0_vector.asDiagonal();
    }
    init_ = true;
    return;
  }
```

The default values of `massLumpingSystem` in our code is set to `False` and we will get more in detail of such choice in section 4.2.2, where we present the tests run to compare the employment of the mass lumping technique, both in R and in C++.

3.4 Changes in the GSR-PDE class

For sake of completeness, here we report some changes operated in the GSR-PDE class.

First we mention a useless redundancy in the definition of `W_` storing the weight matrix. Indeed, in the previous implementation, the GSR-PDE class defined its own private member `W_`, hiding the one of the father class `RegressionBase`. Therefore, when storing the quantities at convergence of FPIRLS, only the private member of GSR-PDE was filled keeping empty the one of `RegressionBase`. To avoid this redundancy we removed the private member of GSR-PDE and enable the employment of the `RegressionBase`'s member properly importing the needed symbols in `ModelMacros.h`. In this way, at the FPIRLS convergence the final weight matrix is only stored in the member `W_` inherited by `RegressionBase`. We also added, in the GSR-PDE `solve()` method, the computation and storing of `XtWX` and `invXtWX` at convergence which were missing but necessary for the GCV computation.

Secondly, we implemented the new methods required by FPIRLS after our generalization, namely:

- `initialize_mu()`: simply returns the vector of observation calling the getter `y()`, since it is the proper initialization for these problems.

- `compute()`: computes the weight and pseudo-observations vectors as follows:

$$\mathbf{pw}\_ = \mathbf{G}^{-2}\mathbf{V}^{-1}$$

$$\mathbf{py}\_ = \mathbf{G}|\mathbf{y} - \boldsymbol{\mu}| + \boldsymbol{\theta}$$

  where $\mathbf{G} = diag(g'(\boldsymbol{\mu}))$ with g canonical link function and $\mathbf{V} = diag(V(\boldsymbol{\mu}))$ with $V(.)$ variance function of the chosen distribution.

- `model_loss()`: computes the unpenalized loss of the model as

$$\texttt{J\_unpenalized} = ||\mathbf{V}^{-\frac{1}{2}}(\mathbf{y} - \boldsymbol{\mu})||^2$$

Finally, we fixed an issue inside the method `norm()` of the class which evaluates the numerator of the GCV scrores. The latter is called inside the method `eval()` in `GCV.h` and receives as inputs the fitted values and the actual observations. Neverthless, in the GSR-PDE framework we cannot directly operate on the fitted values but we need to go back to the fitted mean vector $\boldsymbol{\mu}(\hat{\mathbf{f}}, \hat{\boldsymbol{\beta}})$ given by: $\boldsymbol{\mu}(\hat{\mathbf{f}}, \hat{\boldsymbol{\beta}}) = g^{-1}(\hat{\mathbf{f}}_\mathbf{n} + \mathbf{X}\hat{\boldsymbol{\beta}})$. Indeed, the numerator of the GCV scores is defined as $||\mathbf{y} - \boldsymbol{\mu}(\hat{\mathbf{f}}, \hat{\boldsymbol{\beta}})||_d^2$ where the subscript d indicates the norm induced by the deviance of the chosen distribution. Therefore, instead of passing directly the fitted values to the deviance function it is enough to compute $\boldsymbol{\mu}(\hat{\mathbf{f}}, \hat{\boldsymbol{\beta}})$ through the inverse link function and operate on it as it is shown below.

```
template <typename PDE, typename RegularizationType, typename SamplingDesign,
    typename Solver, typename Distribution>
double GSRPDE<PDE, RegularizationType, SamplingDesign, Solver,  ↩
   Distribution>::norm
(const DMatrix<double>& fitted, const DMatrix<double>& obs) const {
  Distribution distr_{};
  // compute mu
  DMatrix<double> mu = distr_.inv_link(fitted);
  double result = 0;
  for(std::size_t i = 0; i < obs.rows(); ++i)
    result += distr_.deviance(mu.coeff(i,0), obs.coeff(i,0));
  return result;
}
```

In order to test the changes made on the GSR-PDE class we added a test, among the

ones already present in the library, which is discussed in A.0.1. In particular, this test is the first one using the FPIRLS algorithm in the case of areal data and the first one employing the GCV computation in the GSR-PDE framework.

Chapter 4

Tests and results

In this chapter we will first test the implemented SQRPDE model in C++, then all the added functionalities to the library, and finally we will provide a comparison with the R version of the code.

Structure of the tests In the following we will consider n data points located at $\mathbf{p_1}, ..., \mathbf{p_n}$ and generated according to one of the two following strategies:

- Strategy A: heteroscedastic data (with eventual covariates) sampled from a Gaussian with mean and variance function of the space:

$$\mathbf{Y} \sim \mathcal{N}(\mu(x, y), \sigma^2(x, y)) + \mathbf{X}\boldsymbol{\beta} \tag{4.1}$$

- Strategy B: data (with eventual covariates) generated as the sum of a deterministic function of the position and a skewed-normal noise (SN):

$$\mathbf{Y} \sim \boldsymbol{f}(x, y) + \boldsymbol{\eta} + \mathbf{X}\boldsymbol{\beta}, \tag{4.2}$$

  where $\boldsymbol{\eta} \sim S\mathcal{N}(\xi, \omega, \alpha_N)$, with ξ, ω, α_N are parameters to control the skewness of the distribution.

For the sake of completeness we will also cover the areal sampling case, where data are sampled from n subdomains Ω_i following one of the above mentioned strategies. We will consider meshes of N nodes of different geometries and dimensions. In particular, we will test 1D, 2D and 3D meshes embedded into two or three dimensional spaces, covering also the manifolds' case (linear networks (1.5D domains) and surfaces (2.5D domains)). In all cases we will use homogeneous Neumann boundary conditions, since they are the most appropriate for our case, and linear FEM basis functions.

Metrics To assess the quality of the nonparametric part of the results we will obtain, we resort to the Root Mean Square Error (RMSE) which, for each simulation m, is defined as

$$RMSE(m) = \sqrt{\frac{\sum_{j=1}^{N}(\mathbf{f}_j^{[m]} - \hat{\mathbf{f}}_j^{[m]})^2}{N}}, \quad m = 1, ..., M \tag{4.3}$$

where $\mathbf{f}_j^{[m]}$ is the j-th coefficient of the FE representation of the true field f, for the simulation m. In order to have a measure of the goodness of the fit in each sample location $\mathbf{p}_i$, we also take into account the Spatial Root Mean Square Error (SpRMSE)

$$SpRMSE(\mathbf{p}_i) = \sqrt{\frac{\sum_{m=1}^{M}(\mathbf{f}(\mathbf{p}_i) - \hat{\mathbf{f}}_m(\mathbf{p}_i))^2}{M}}, \quad i = 1, ..., n \tag{4.4}$$

For what concerns the parametric part, we will simply show the distribution over M simulations of the estimated coefficients $\hat{\boldsymbol{\beta}}$, in case there are some covariates.

4.1 Simulation studies

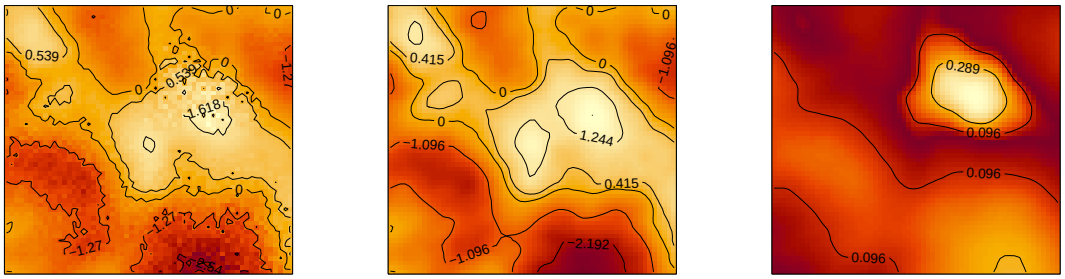
Given the setting specified in 4, we wrap up the C++ tests in the following table

Domain dimension	Domain shape	Covariates	Sampling design	Penalty	Test
2D	Unit square	no	pointwise at nodes	Δf	1
	Unit square	no	pointwise at nodes	$K\nabla f$	2
	C-shaped	yes	areal	Δf	3
3D	Unit sphere	yes	pointwise at locations	Δf	4
1.5D	C-shaped network	no	pointwise at nodes	Δf	5
2.5D	Hub	yes	pointwise at nodes	Δf	6

In those tests, we have chosen the optimum value of λ using the exact GCV strategy.

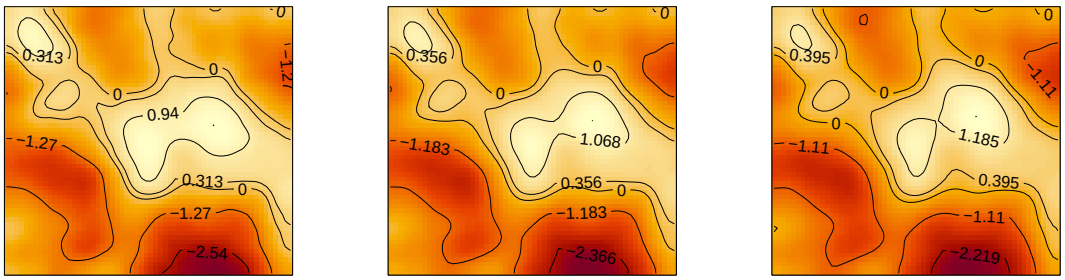
4.1.1 Test 1: Nonparametric 2D Unit square domain

For the first test we start from the simplest case, namely a unit square domain where data have no covariates, selecting the laplacian as penaltization term. The sampling is pointwise at the mesh nodes and it follows the strategy A described in 4. In particular, the mean and the standard deviation are gaussian random fields generated with the R function *grf* from the package *geoR*. The data, the correspondent mean and standard deviation fields are shown in 4.1. The results of the simulation over a mesh of 3600 nodes and 10 simulations are shown in 4.2 and in 4.3, for different levels of the quantile order α, specifically $\alpha = (10\%, 25\%, 50\%, 75\%, 90\%)$.

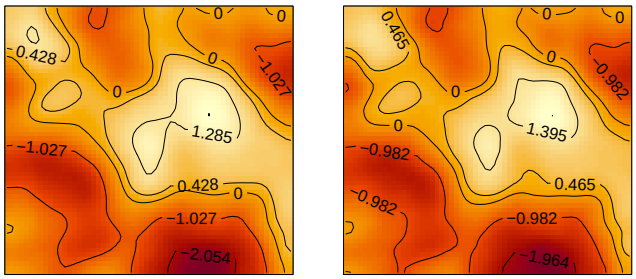


(a) Data (b) Mean $\mu(x, y)$ (c) Std deviation $\sigma(x, y)$

Figure 4.1: Test 1: Simulation settings with $n = N = 3600$. Observations are shown as a continuous interpolation of their discrete values.

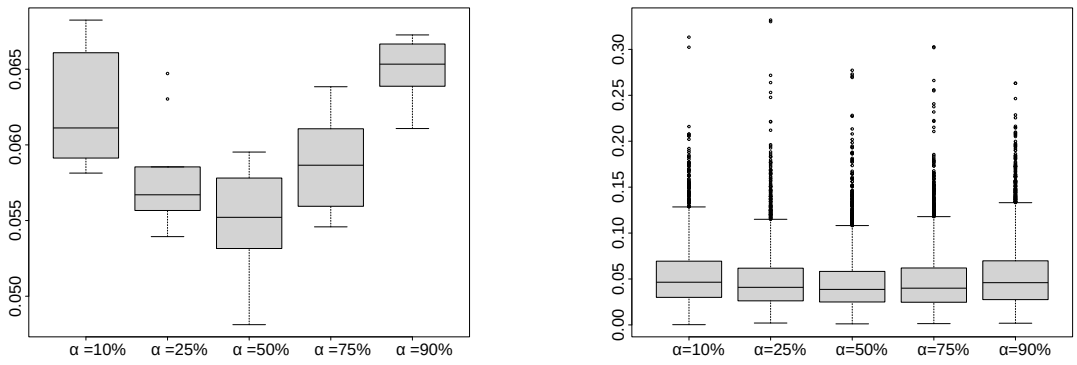


(a) $\alpha = 10\%$ (b) $\alpha = 25\%$ (c) $\alpha = 50\%$



(d) $\alpha = 75\%$ (e) $\alpha = 90\%$

Figure 4.2: Test 1: Fitted fields for different levels of α, for a fixed simulation. We can observe that, as the quantile level increases, the correspondent fitted surface presents more and more spikes in those regions manifesting high values of the data.



(a) RMSE

(b) SpRMSE

Figure 4.3: Test 1: Accuracy results for different levels of α. From the left panel, we can see that, for all the α_s, the range of the RMSE is lower than the one of the standard deviation function shown above, meaning that the model is capable to capture the true spatial field for any level of quantile order. Moreover the RMSE reaches the lowest values while estimating the median and slightly increases moving to more extreme quantiles. The SpRMSE shows a larger range due to the presence of outliers, mainly corresponding to nodes on the border of the domain since the Neumann boundary conditions are not met analitically. Neverthless, it is still inside the range of the standard deviation function.

4.1.2 Test 2: constant PDE coefficients 2D Unit square domain

Here we want to test the case with a non-trivial penalty term. We generate data according to the strategy B, with a pointwise sampling at the mesh nodes. In particular, the deterministic field is given by

$$f(x, y) = sin(2\pi(x + 0.17))cos(2\pi y) + 4sin(3\pi(x + 0.17)) \tag{4.5}$$

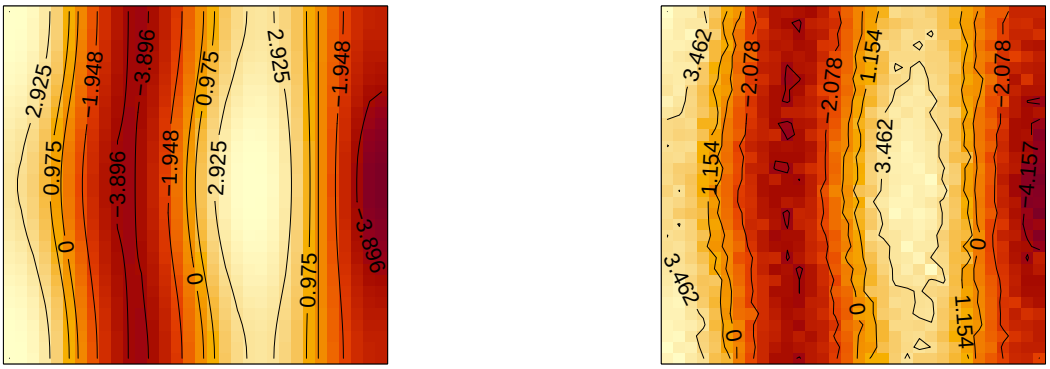
where the translation along the x-axis is imposed to almost fulfill the Neumann boundary conditions. Then we add a skewed normal noise $\boldsymbol{\eta} \sim SN(\xi, \omega, \alpha_N)$, with $\xi = 4, \omega = 0.05\ range(f), \alpha_N = 5$

The smooth deterministic field f and the data are plotted in 4.4. Looking at the data pattern, we can see that there seems to be some anisotropic effect in the vertical direction. For this reason we decide to use a diffusion penalty term, with diffusion matrix given by

$$\mathbf{K} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix} \tag{4.6}$$

to encode the diffusion effect along that direction. The test is repeated over 10 simulations for three quantile orders: $\alpha = (10\%, 50\%, 90\%)$. The fitted fields are shown in

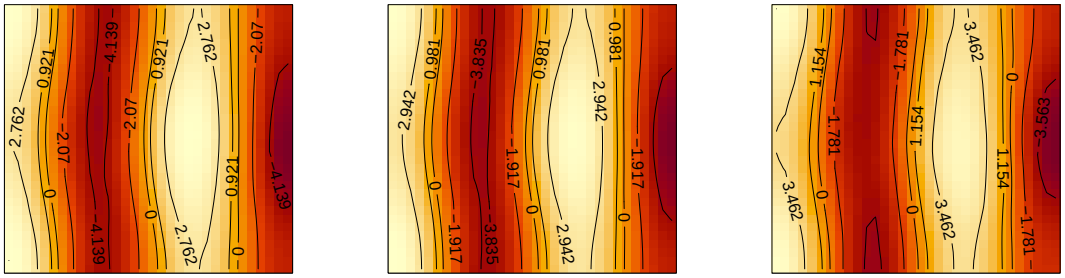
figure 4.5 and the accuracy results in figure 4.6.



(a) Spatial field $f(x, y)$: high values yellow, small values red

(b) Data: high values yellow, small values red

Figure 4.4: Test 2: Simulation settings with $n = N = 1024$. Observations are shown as a continuous interpolation of their discrete values.

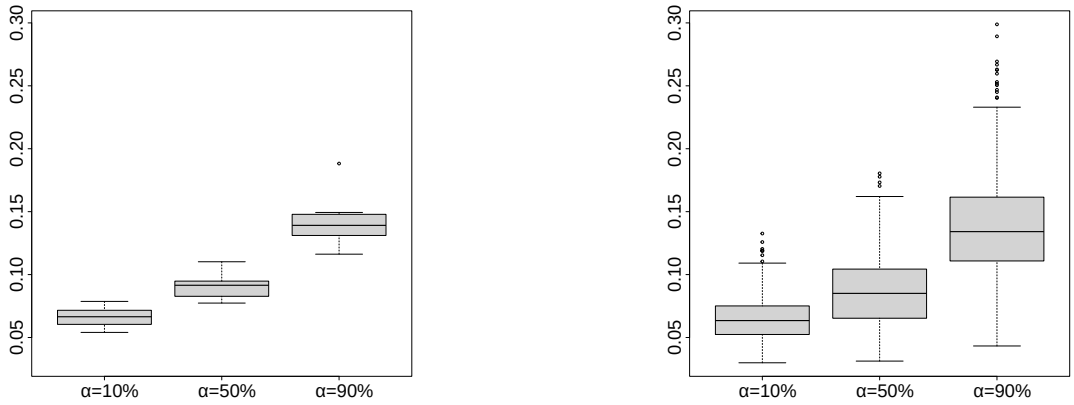


(a) $\alpha = 10\%$

(b) $\alpha = 50\%$

(c) $\alpha = 90\%$

Figure 4.5: Test 2: Fitted fields for $\alpha = (10\%, 50\%, 90\%)$. We can observe that all the fitted fields show a smooth behaviour and capture the anisotropic effect present in the data.



(a) RMSE

(b) SpRMSE

Figure 4.6: Test 2: Accuracy results for $\alpha = (10\%, 50\%, 90\%)$. The RMSE and the SpRMSE are below the noise standard deviation, which is around 0.4, for all α_s, meaning that the fitted fields are well approximating the true ones as expected from their representation. Moreover, we can notice that the errors for $\alpha = 90\%$ are larger due to the fact that the noise is mainly affecting the higher data values, keeping almost unchanged the lower ones.

4.1.3 Test 3: Areal sampling 2D C-shaped domain

Here we present a simulation study for areal data. The data are collected over $n = 20$ subdomains defined in a C-shaped domain. Each domain D_i is constituted by two adjacent triangles of the mesh as displayed in figure 4.7b. Let $a(x, y)$ and $d(x, y)$ be the following functions:

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

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

then our spatial field is given by:

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

and the data are generated as follows:

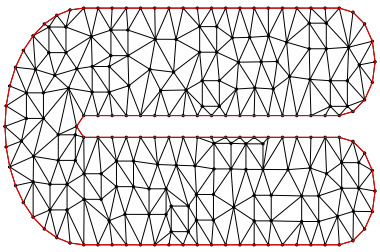
$$y_i = x_i\beta + \frac{1}{|D_i|}\int_{D_i} f + \eta_i, \quad i = 1, ..., n \tag{4.7}$$

where $\beta = 3$, $\mathbf{x} \sim Beta(2, 2)$ is the covariate vector and $\boldsymbol{\eta} \sim SN(\xi, \omega, \alpha_N)$ is the noise vector. The simulation settings are shown in figure 4.7. We consider a simple laplacian penalization and the smoothing parameter λ is chosen through the exact GCV computation.

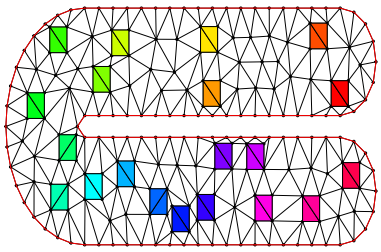
Since we are now dealing with areal observations we need to redefine the $SpRMSE$ as follows:

$$SpRMSE(D_i) = \sqrt{\frac{1}{M}\sum_{m=1}^{M}\left(\frac{1}{|D_i|}\int_{D_i}\hat{f} - \frac{1}{|D_i|}\int_{D_i} f\right)^2}, \quad i = 1, ..., n \tag{4.8}$$

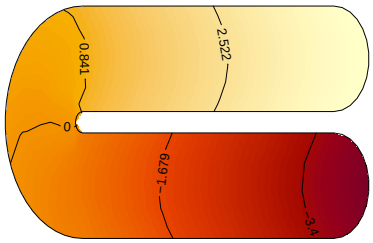
The test is repeated over 10 simulations for three quantile orders: $\alpha = (10\%, 50\%, 90\%)$. The fitted fields are shown in figure 4.8 and the accuracy results in figure 4.9.



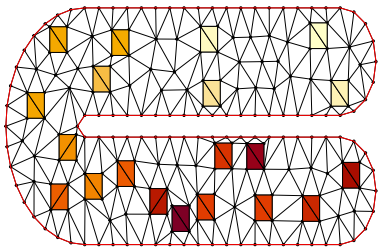
(a) C-shaped mesh



(b) Mesh with subdomains

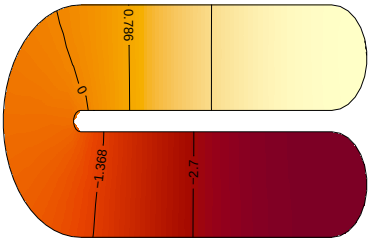


(c) Spatial field $f(x, y)$: high values yellow, small values red

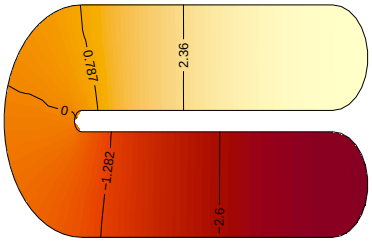


(d) Samples over regions: high values yellow, small values red

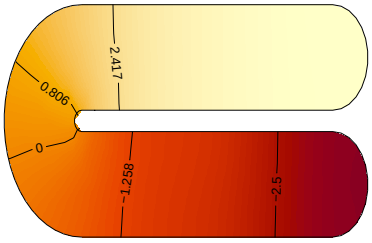
Figure 4.7: Test 3: Simulation settings for areal data. Subdomains are shown in the right top panel and highlighted in different colours: each subdomain D_i is a square composed by 2 triangles of the mesh. The spatial field f is shown in the bottom part pf the figure alongside with the sampled values over the regions.



(a) $\alpha = 10\%$

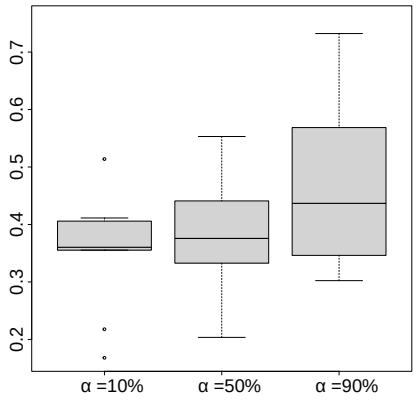


(b) $\alpha = 50\%$

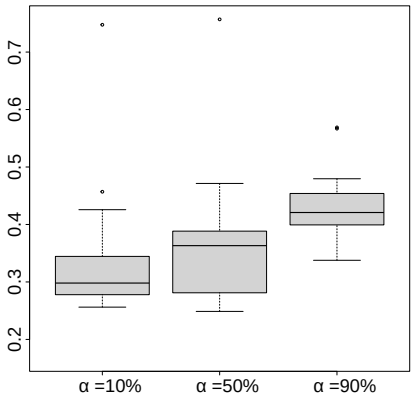


(c) $\alpha = 90\%$

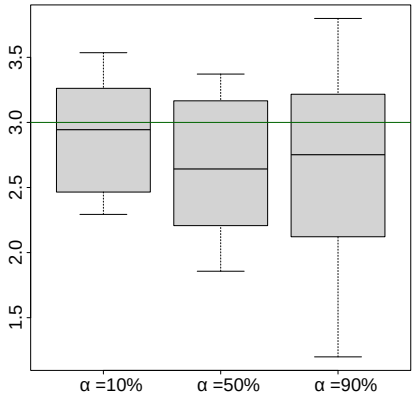
Figure 4.8: Test 3: Fitted fields for $\alpha = (10\%, 50\%, 90\%)$. The pattern of the fitted fields is the one expected from the definition of $f(x, y)$ and we can observe as the magnitude of the contour lines increases moving towards higher quantile levels.



(a) $RMSE$



(b) $SpRMSE$



(c) Boxplot of $\hat{\beta}$ distribution around true value (darkgreen line)

Figure 4.9: Test 3: Accuracy results $\alpha = (10\%, 50\%, 90\%)$. The RMSE and SpRMSE are comparable with respect to the noise standard deviation which is around 0.5. They are also comparable among the different α_s even though the case with $\alpha = 90\%$ is associated with higher values of both metrics. The true value of β is contained in the $\hat{\beta}$ boxplots for all the analyzed quantile levels.

4.1.4 Test 4: Semiparametric 3D Unit sphere domain

We now present a test over a 3D unit sphere domain. The locations are sampled randomly within the sphere and we consider $n = 2000$ data observations. Data are generated according to strategy A described in 4. The mean function is defined as:

$$\mu(x, y) = sin(2\pi x) + sin(2\pi y) + 3sin(2\pi z)$$

and the standard deviation function is radial following the definition below:

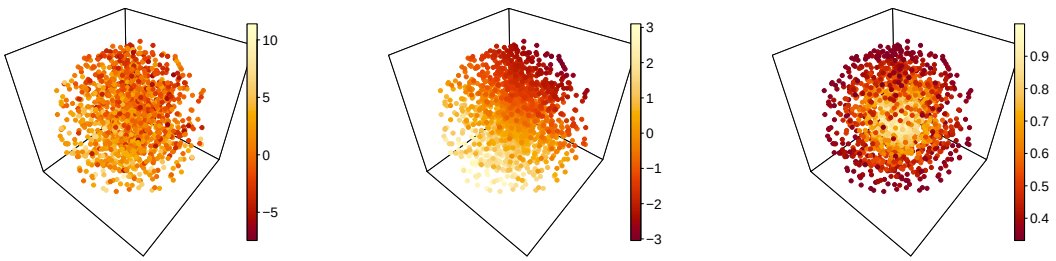
$$\sigma(x, y) = \sigma(\rho) = \frac{1}{\sqrt{\rho} + 0.1}$$

with $\rho = \sqrt{x^2 + y^2 + z^2}$.

The covariates are generated as

$$X_{1,i} \sim \mathcal{N}(0, 1), \;\; X_{2,i} \sim \mathcal{E}(1) \;\; \forall i = 1, ..., n$$

with $\boldsymbol{\beta} = (2, 1)^T$. We apply a laplacian penalization and the smoothing parameter λ is chosen through the exact GCV computation. The data and the correspondent mean and standard deviation fields are shown in figure 4.10. The test is run for the quantile orders $\alpha = (10\%, 50\%, 90\%)$ and repeated over 10 simulations. The fitted fields are displayed in figure 4.11 and the accuracy results are shown below in figure 4.12.



(a) Data (b) Mean $\mu(x, y)$ (c) Std deviation $\sigma(x, y)$

Figure 4.10: Test 4: Simulation settings for heteroscedastic data over a 3D sphere domain. The data, the mean and the standard deviation fields are shown and coloured according to their magnitude.

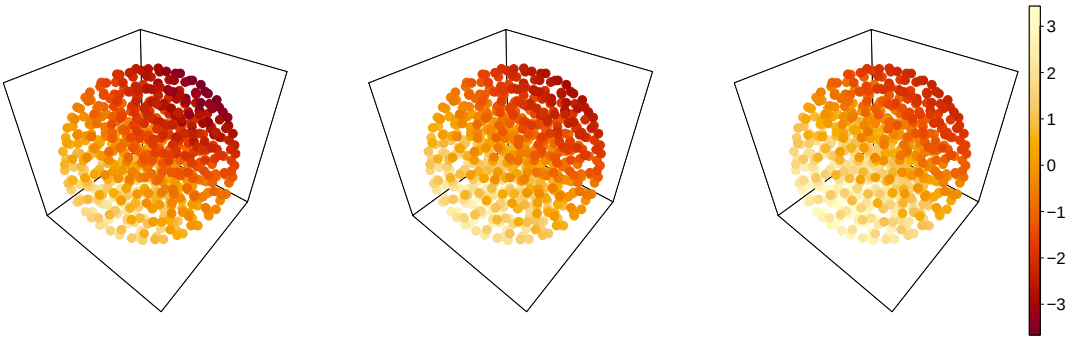
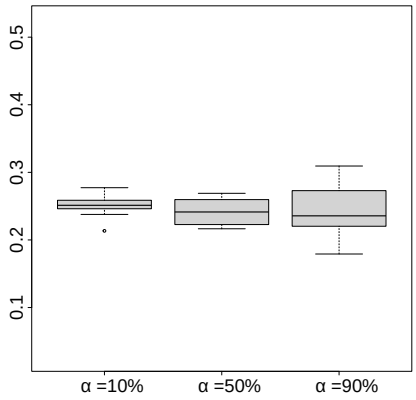
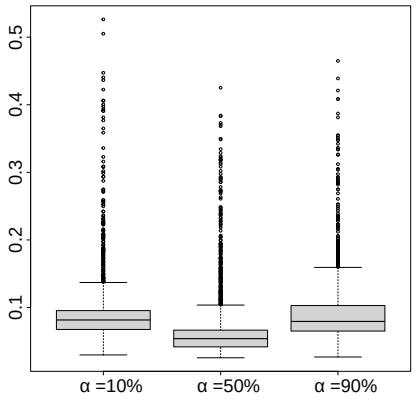


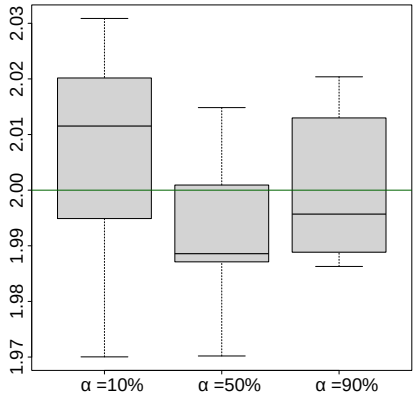
Figure 4.11: Test 4: Fitted fields for $\alpha = (10\%, 50\%, 90\%)$ read from left to right. The fields are displayed with the same color scale shown aside, therefore we can notice as the field becomes lighter, meaning larger, moving towards higher quantile orders.



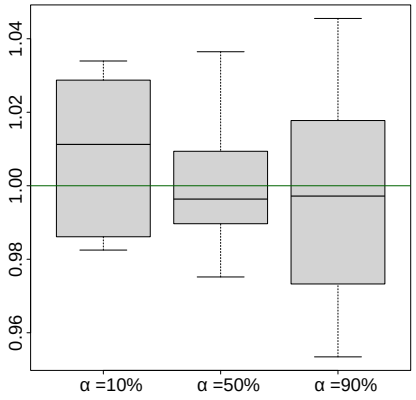
(a) $RMSE$



(b) $SpRMSE$



(c) Boxplot of $\hat{\beta}_1$ distribution around true value (darkgreen line)



(d) Boxplot of $\hat{\beta}_2$ distribution around true value (darkgreen line)

Figure 4.12: Test 4: Accuracy results for $\alpha = (10\%, 50\%, 90\%)$. For all the analyzed level of α the RMSE and the SpRMSE are good compared with the range of the standard deviation function used to generate the data. The boxplots contain the true value of both β_1 and β_2 in all cases and they are in a narrow range.

4.1.5 Test 5: Nonparametric 1.5D C-shaped network domain

Here we present a test on a linear C-shaped network. The setting is very simple, with data generated as

$$\mathbf{Y} \sim f(x, y) + \mathcal{N}(0, \sigma) \tag{4.9}$$

with $\sigma = 0.1\ range(f)$ and f which is a combination of trigonometric functions, shown in 4.13, together with the data, scattered at the mesh nodes. The penalization is a simple laplacian and the optimal parameter λ is selected through the exact GCV computation. The fitted fields for a fixed simulation are displayed in 4.14 and the accuracy results over 10 simulations in 4.15 for the quantile orders $\alpha = (10\%, 50\%, 90\%)$.

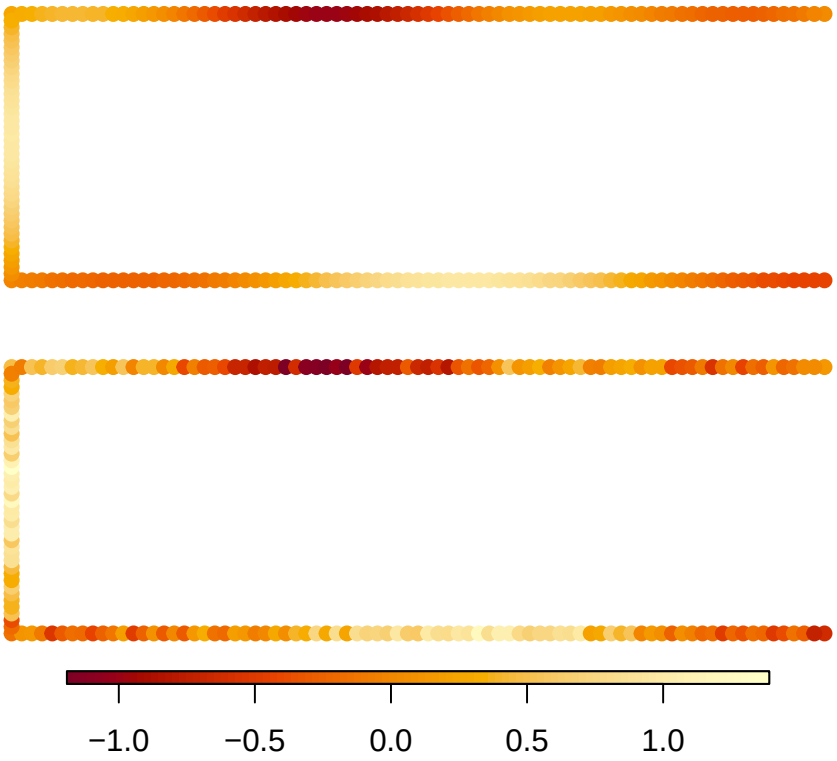


Figure 4.13: Test 5: Simulation settings for C-shaped linear network. Smooth deterministic field f (on top) scattered at the mesh nodes; data points (on the bottom) with $n = N = 201$

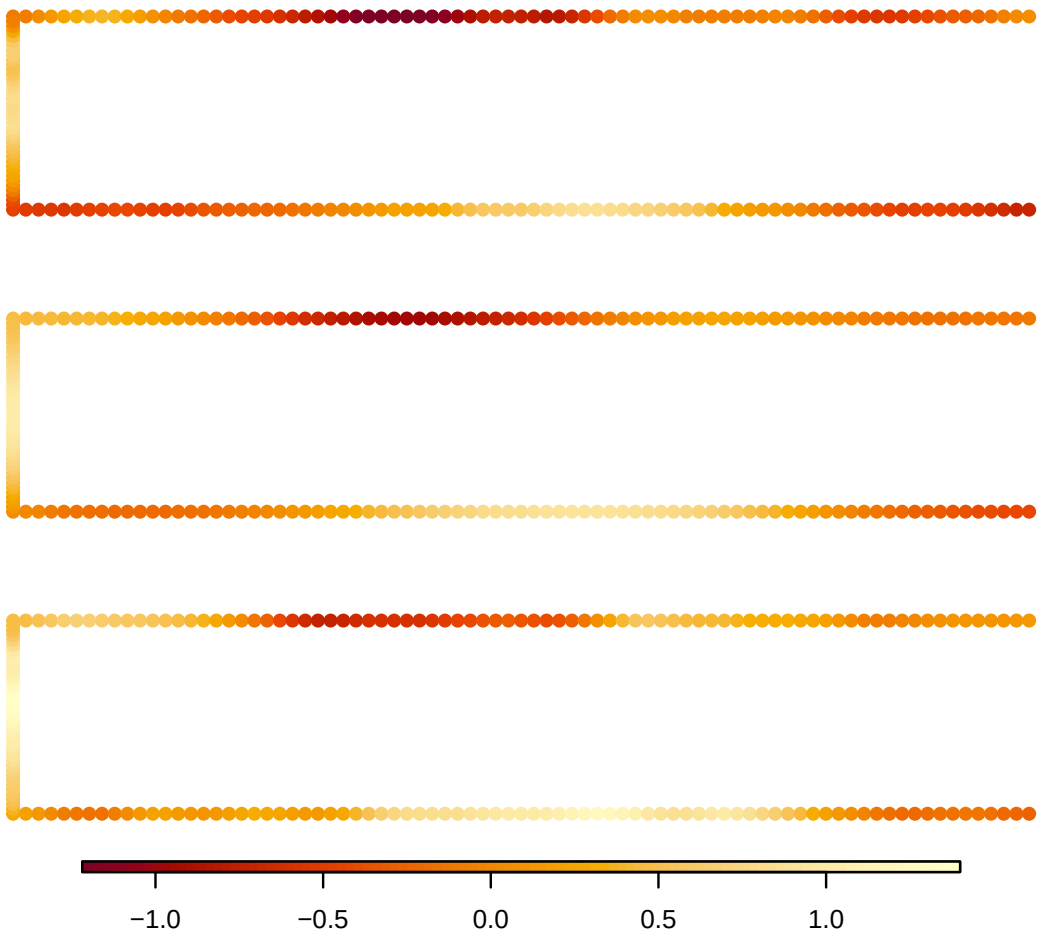
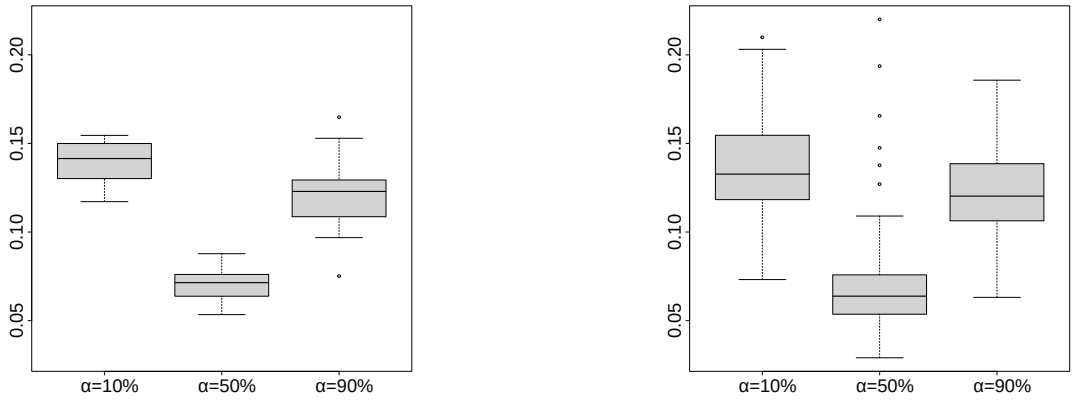


Figure 4.14: Test 5: Fitted fields for $\alpha = (10\%, 50\%, 90\%)$ displayed from the top to the bottom. The fitted fields show the same pattern present in the deterministic field $f(x, y)$ and we can observe as they become larger moving towards higher quantile orders.



(a) $RMSE$ (b) $SpRMSE$

Figure 4.15: Test 5: Accuracy results for $\alpha = (10\%, 50\%, 90\%)$. The test analyzing the median shows better results, this can be due to the fact that the data are generated adding a normal noise, therefore the tails are not so well represented. Yet the results are good for all the α_s since they are always below the noise standard deviation which is around 0.2.

4.1.6 Test 6: Semiparametric 2.5D Hub domain

In this test we model a complex domain, noted as Hub domain, that is a 2D surface embedded in a 3D space. The sampling is pointwise at the mesh nodes with $n = 1296$. Data are generated according to strategy A described in 4. The mean and the standard deviation functions are defined as follows:

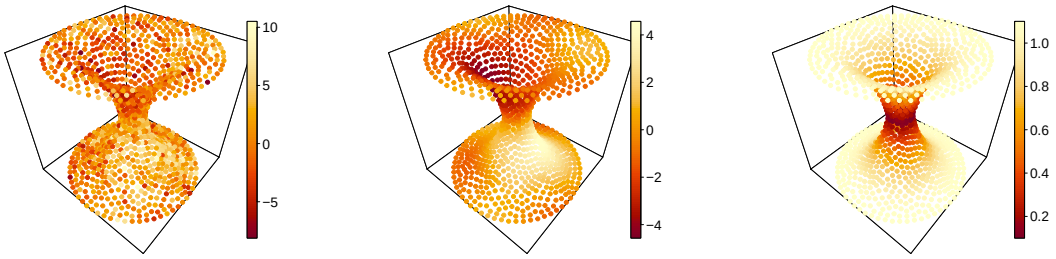
$$\mu(x, y) = sin(2\pi x) + sin(2\pi y) + 3sin(2\pi z)$$

$$\sigma(x, y) = 2|z - 0.5| + 0.1$$

The covariates are generated as

$$X_{1,i} \sim \mathcal{N}(0, 1), \;\; X_{2,i} \sim \mathcal{E}(1) \;\; \forall i = 1, ..., n$$

with $\boldsymbol{\beta} = (2, 1)^T$. The smoothing parameter λ is chosen through the exact GCV computation and a simple laplacian penalization is applied. The data and the correspondent mean and standard deviation fields are shown in figure 4.16. We analyze the quantile orders $\alpha = (10\%, 50\%, 90\%)$ and the test is repeated over 10 simulations. The fitted fields for a fixed simulation are shown in figure 4.17 and the accuracy results are below in figure 4.18.



(a) Data (b) Mean $\mu(x, y)$ (c) Std deviation $\sigma(x, y)$

Figure 4.16: Test 6: Simulations settings over a 2.5D Hub domain. The data, the mean and the standard deviation fields are shown and coloured according to their magnitude.

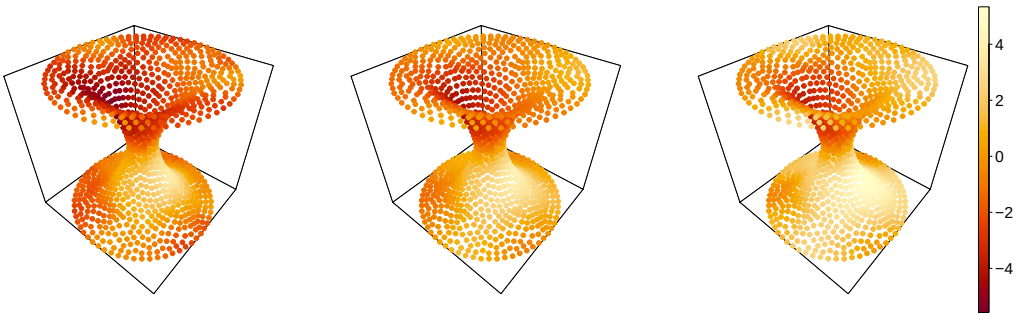
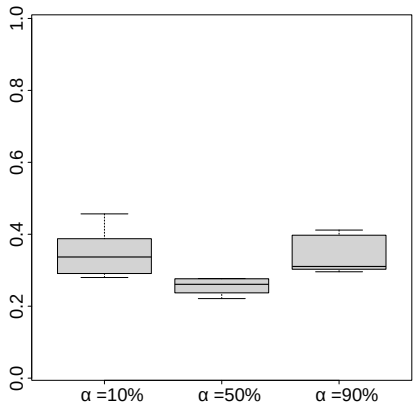
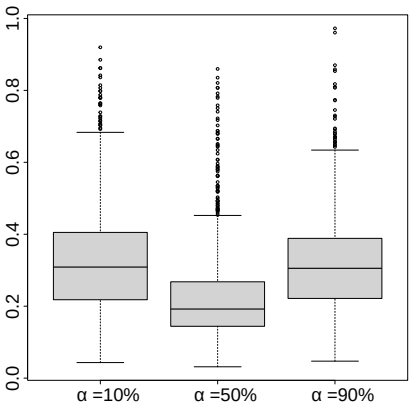


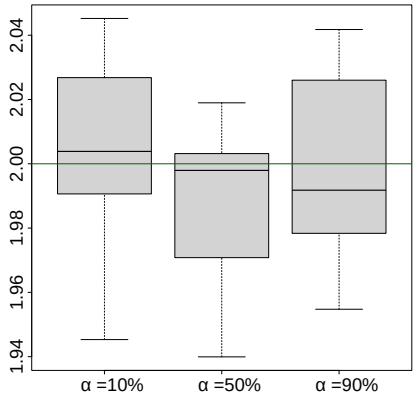
Figure 4.17: Test 6: Fitted fields for $\alpha = (10\%, 50\%, 90\%)$. The fitted fields show a smooth behaviour and they become correctly larger moving towards higher quantile orders.



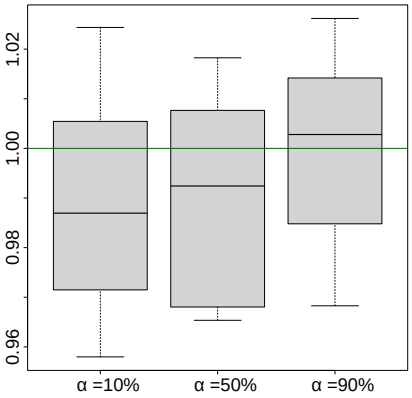
(a) $RMSE$



(b) $SpRMSE$



(c) Boxplot of $\hat{\beta}_1$ distribution around true value (darkgreen line)



(d) Boxplot of $\hat{\beta}_2$ distribution around true value (darkgreen line)

Figure 4.18: Test 6: Accuracy results for $\alpha = (10\%, 50\%, 90\%)$. The RMSE and the SpRMSE are inside the range of the standard deviation function used to generate the data and in particular they are always lower of its maximum value for all the levels of α. The boxplots of $\hat{\beta}_1$ and $\hat{\beta}_2$ always contain the true value for all the α and they have a narrow range.

4.2 New functionalities

The goal of this section is to test the new functionalites added to the `fdaPDE` library. In particular, we want to compare the performances of two competing solvers, Woodbury and Cholesky, both in terms of computational time and memory consumption. Finally, we propose a test to check whether the new mass lumping functionality brings an effective time improvement in the C++ code, showing at the same time how beneficial was this option for the R version of the code.

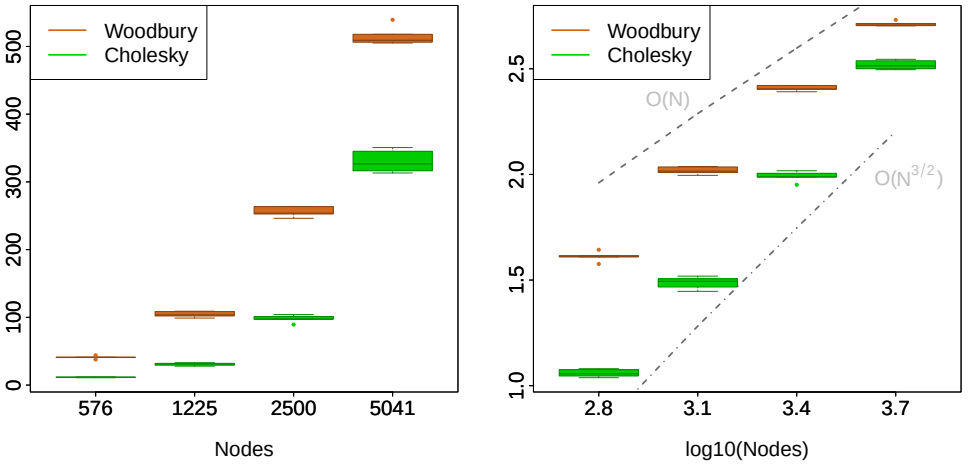
4.2.1 Test 7: stochastic trace approximation

In order to compare the Woodbury and the Cholesky solver for the stochastic trace approximation of the smoothing matrix we conduct two tests over two diverging sequences of mesh nodes, to assess which solver is more efficient in terms of time and memory consumption. The tests are run with $\alpha = 50\%$ and we fix the number of Monte Carlo simulations to 1000. The specifications of the two tests are shown in the table below:

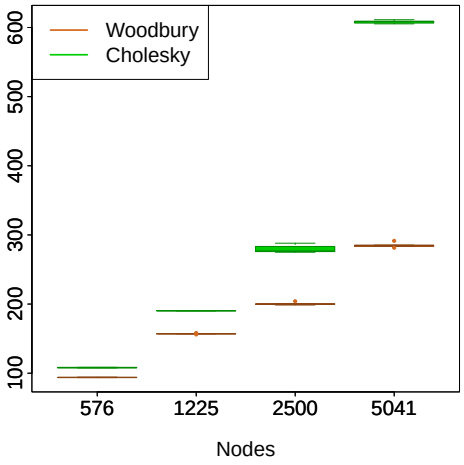
Domain	Data	Quantile order α	CPU	Test
Unit square	skewed	0.5	*Intel(R) Core(TM) i5-8250U*	7.1
C-shaped	heteroscedastic	0.5	*Intel(R) Core(TM) i7-1065G7*	7.2

Table 4.1: Settings of the two tests run for the comparison between Woodbury and Cholesky solver for the stochastic trace approximation.

Test 7.1 and 7.2 As presented in table 4.1, the first test is on a squared domain with data given by the sum of a smooth deterministic function, a parametric contribution and a skewed random noise. It is run on an *Intel(R) Core(TM) i5-8250U* with 8 GB of installed RAM. Instead, the second test is conducted over a C-shaped domain with heteroscedastic semiparametric data and it is run on an *Intel(R) Core(TM) i7-1065G7* processor with 8 GB of RAM. The data settings are shown more specifically in A.0.3 for Test 7.1 and in A.0.2 for Test 7.2. The chosen sequence of mesh nodes is $N = [576, 1225, 2500, 5041, 10000, 20164, 40000, 80089]$ with fixed sample size $n = 3969$ for Test 7.1 and $N = [742, 1452, 2789, 5502, 11011, 22086, 44136, 85169]$ with fixed $n = 2789$ for Test 7.2. The memory consumption is measured in terms of the Resident Set Size (RSS) used by the two algorithms. The results are shown in figure 4.19 and 4.20 for the two tests respectively.

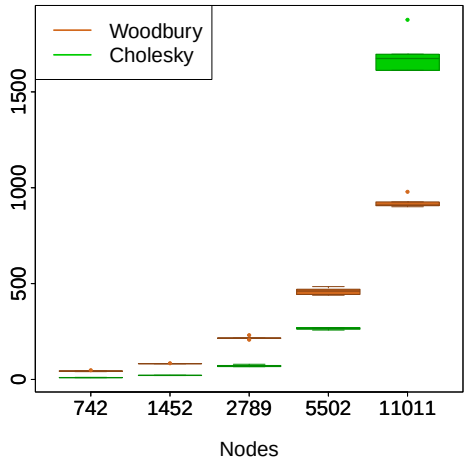


(a) Computational time [seconds] (b) Logarithmic scale in time [seconds]

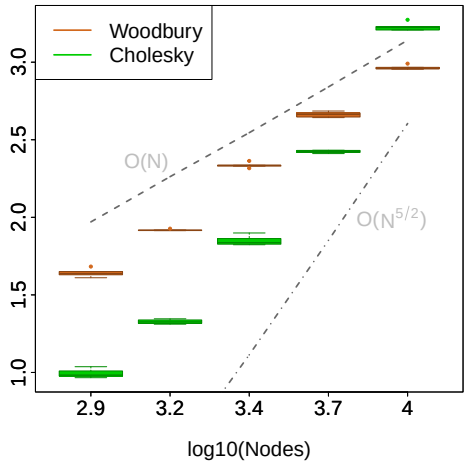


(c) Memory consumption [Mb]

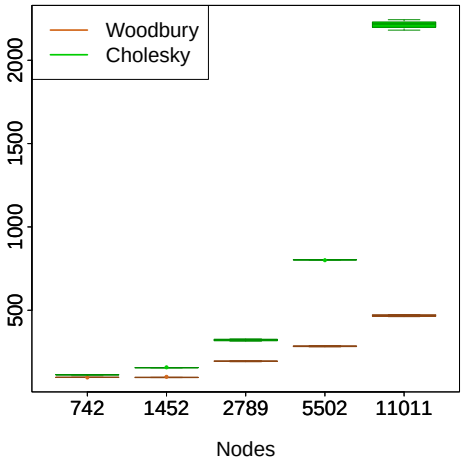
Figure 4.19: Test 7.1 skewed data over squared domain: comparison between Woodbury and Cholesky algorithms in terms of computational time and memory effort. The results are shown up to $N = 5041$ since for denser meshes the Cholesky algorithm crashes due to the memory overflow; instead the Woodbury algorithm is able to perform the tests for all cases except the last one $N = 80089$.



(a) Computational time [seconds]



(b) Logarithmic scale in time [seconds]



(c) Memory consumption [Mb]

Figure 4.20: Test 7.2 heteroscedastic data over C-shaped domain: comparison between Woodbury and Cholesky algorithms in terms of computational time and memory effort. The results are shown up to $N = 11011$ since for denser meshes the Cholesky algorithm fails due to the memory overflow (as in Test 7.1); instead the Woodbury algorithm is able to perform the tests for all cases except the last one $N = 85169$.

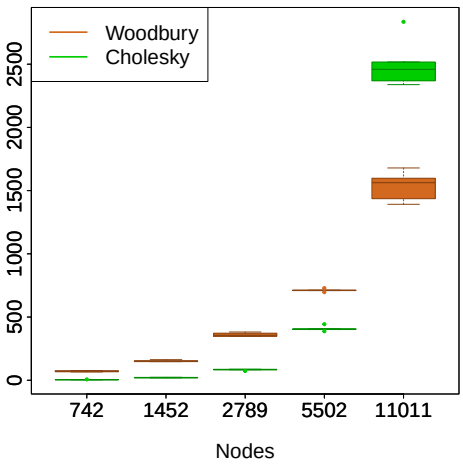
From the memory point of view we can observe that the Woodbury algorithm has a reduced RSS usage compared to the Cholesky one. In particular, this discrepancy is not so significant for a limited number of nodes, approximately up to $N = 3000$, and becomes much more evident for denser meshes. Focusing on the computational time instead, the Cholesky factorization is faster up to around $N = 5000$ and then either crashes, as in Test 7.1, either becomes slower than Woodbury, as in Test 7.2. Indeed, from figure 4.19 and 4.20, we can observe that Cholesky spends a low amount of time for meshes with limited number of nodes and then rapidly grows showing a more than

linear order in time (in particular in Test 7.2 we have a more than quadratic growth) against the linear dependence shown by Woodbury.

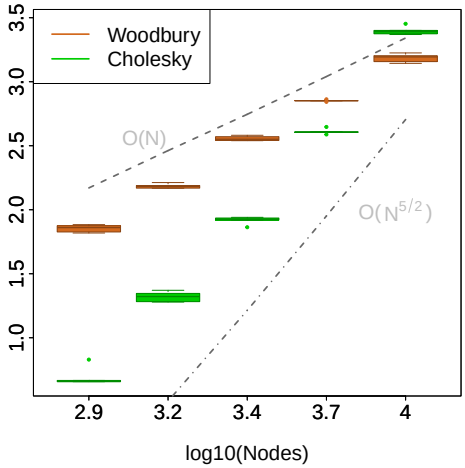
Therefore, since our main goal is to achieve the greatest efficiency possible, we state that the Cholesky algorithm performs better than Woodbury for relative small systems being faster in spite of a negligible increase in the memory consumption. Instead, for larger systems the Woodbury algorithm is preferable since it is less demanding both in time and memory.

The goal now is to set an appropriate threshold so that the compiler can automatically choose which algorithm to apply. For the discussed reasoning we believe that a proper choice for such threshold is $N_{threshold} = 3000\ Nodes$ which has shown, in both tests, a good balance between the memory usage and the time speed-up.

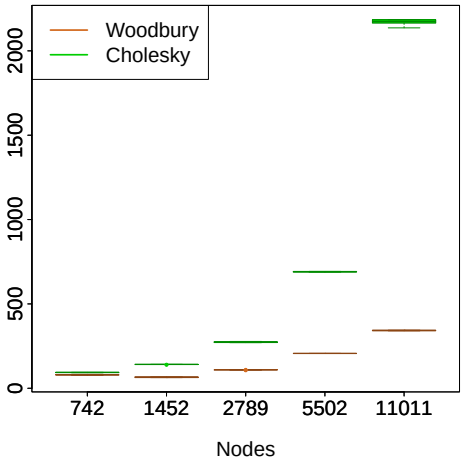
Test SR-PDE In conclusion, since such choice for the $N_{threshold}$ will affect also the other statistical models inside the `fdaPDE` library, we implement a test for a SR-PDE model in order to check the validity of our new implementation also in this setting. Specifically, the test is the same performed for the SQR-PDE case over a C-shaped domain; the analyzed sequence of mesh nodes is $N = [742, 1452, 2789, 5502, 11011]$ and the number of Monte Carlo simulations is still set to 1000. As we can observe from figure 4.21 the behaviour is the same shown in the two previous tests both for the memory consumption and the computational time. Therefore we can draw the same conclusion discussed above validating our choice for the $N_{threshold}$.



(a) Computational time [seconds]



(b) Logarithmic scale in time [seconds]



(c) Memory consumption [Mb]

Figure 4.21: SR-PDE test over C-shaped domain: comparison between Woodbury and Cholesky algorithms in terms of computational time and memory effort.

4.2.2 Test 8: mass lumping implementation

Once the implementation of the mass lumping option is available on both R and C++ software as described in 3.3.2, we conduct a nonparametric test over a square domain to assess whether there is a difference in terms of computational time and accuracy. The data are generated following strategy B in 4 and shown in the appendix section A.0.4. Two strategies are compared:

- `lumpFALSE`: `massLumpingSystem=False`
- `lumpTRUE`: `massLumpingSystem=True`

The test is repeated over 10 simulation for the quantile order $\alpha = 50\%$ leading to the results shown below.

The computational time is shown in figure 4.23 for both software. We can notice that in R the speed up is significant when the mass lumping technique is active. Instead in C++, the computational time is approximately constant among the two strategies.

From figure 4.22, we can analyze the RMSE as function of λ activating or not the mass lumping option. The lambda sequences are obtained by subsequent refinement over the region minimizing the RMSE so that all the lambdas can be considered to be optimal. Following this approach, we can select lambda as the mean of the sequence and use it for the time performance test. Since the test is nonparametric, the algorithm of the two codes coincides. Therefore, concerning the accuracy, we choose to plot only the C++ results. The RMSE obtained without mass lumping is slightly lower, which is expected since we are not applying any approximation.

In conclusion, in R it is necessary to apply the mass lumping technique to reduce the computational effort, indeed the time benefit is significant especially when the problem size increases. For this reason, this is the only case tested in previous analysis by C.Castiglione. Instead, in C++ such procedure does not lead to a significant improvement from a computational time point of view. Therefore the default values of `massLumpingSystem` in our code is set to `False` to avoid unnecessary approximations. We anyway decided to keep such implementation to allow the user to change this option if needed.

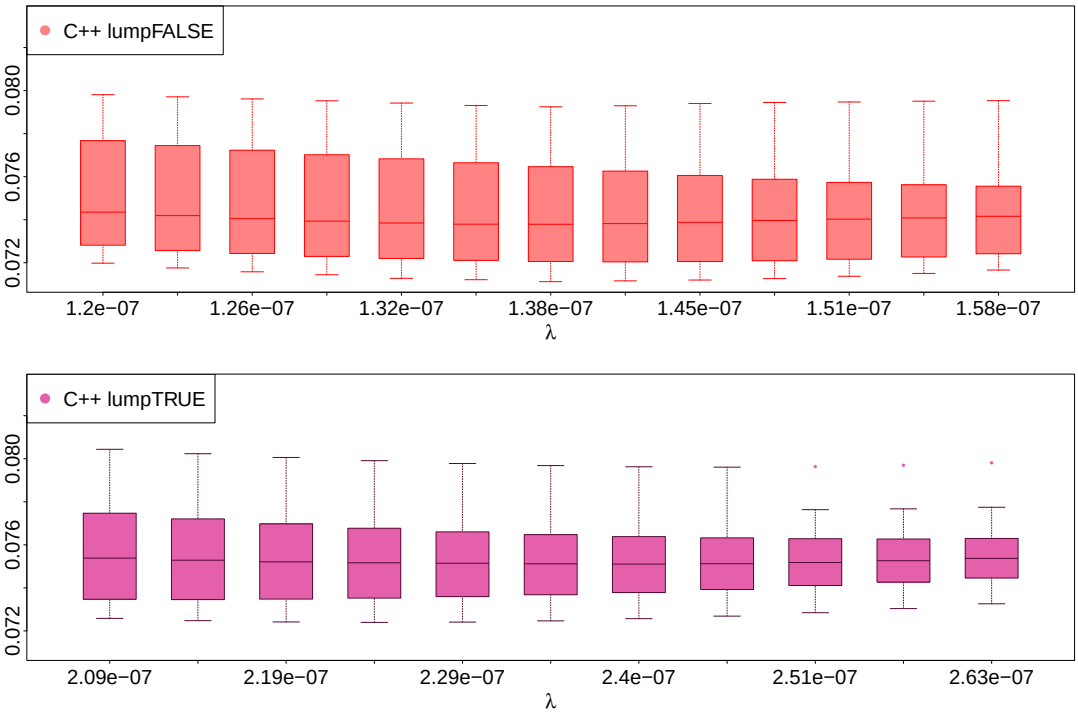


Figure 4.22: Comparison test for the mass lumping option: RMSE curve as function of the optimal lambda sequence. On the top panel, the mass lumping option is disactivated, on the bottom one is activated. Being the sequences optimal, the values of the boxplots are all in the same range.

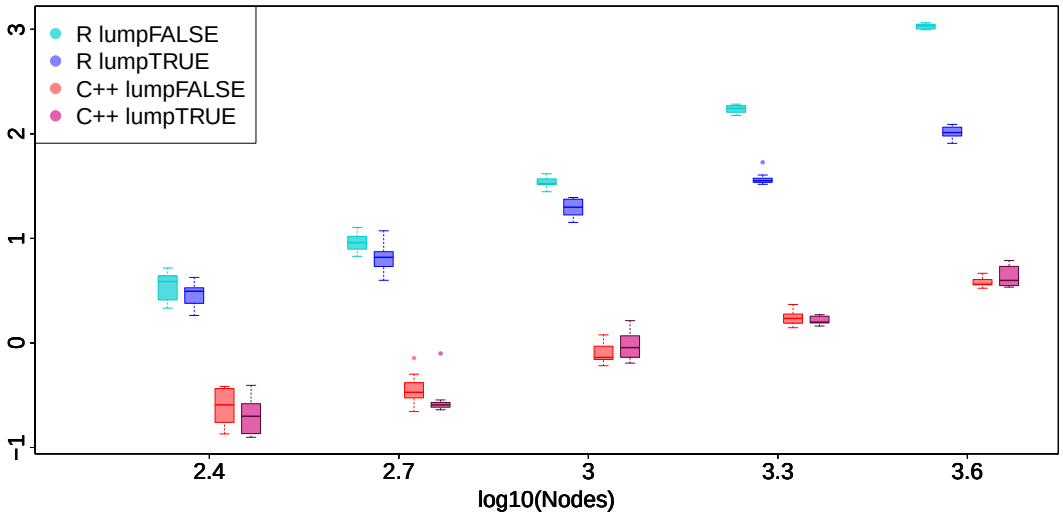


Figure 4.23: Comparison test for the mass lumping option: Computational time (in logaritmic scale) over a diverging sequence of mesh nodes. Four methods are compared: R lumpFALSE (in cyan), R lumpTRUE (in blue), C++ lumpFALSE (in red), C++ lumpTRUE (in violet). We can observe as the R curves are both above the C++ ones. In particular, the R lumpFALSE one is significantly higher than R lumpTRUE. Instead, there is no appreciable difference in magnitude between the C++ results.

4.3 R/C++ comparison

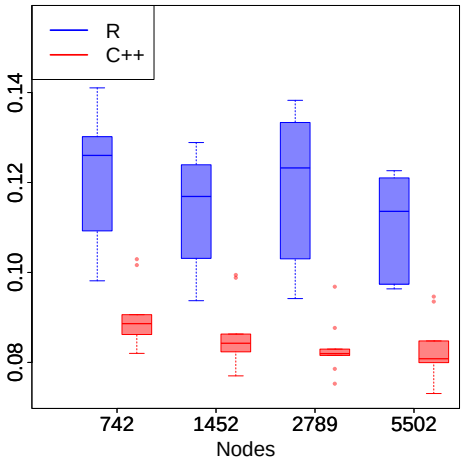
Here we will provide a comparison between the C++ and the R version of the code both in terms of performance and accuracy of the results. We will show semiparametric tests only, since this is the case where the two algorithms differ and eventual differences in terms of accuracy of the solution can be spotted. At this reguard, we recall that R implements an iterative algorithm with a component-wise approach, while in C++ the solution (f, β) is found simultaneously at each step of the FPIRLS algorithm. Moreover, for the R implementation the mass lumping option is activated to guarantee time efficiency as discussed in section 4.2.2, instead for the C++ test we keep the default strategy without applying the mass lumping technique. For two tests the algorithms are compared in terms of accuracy, analizing the RMSE and the fitted $\boldsymbol{\beta}$, and in terms of computational time. In the first test we will analyze a diverging sequence of mesh nodes, keeping fixed the sample size, while in the second one we fix the mesh size and explore a sequence of sample size values. For both tests, the GCV strategy is the exact one and the penalty is the simple laplacian.

4.3.1 Test 9: Semiparametric 2D C-shaped domain

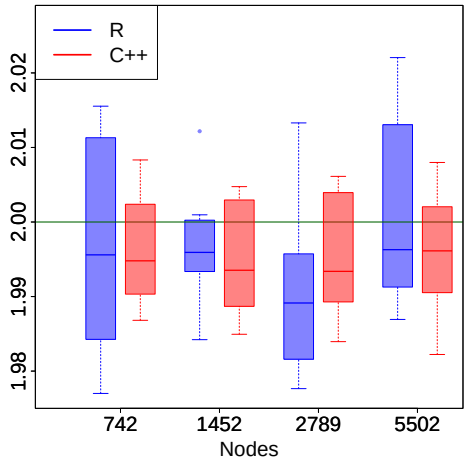
The first comparison test follows the same setting of Test 7.2 described in table 4.1. More precisely, the simulation is conducted over a C-shaped mesh with divergent number of nodes given by the sequence $N = [742, 1452, 2789, 5502]$, with fixed sample size $n = 2789$. Data are generated according to strategy A in 4, where the mean and standard deviation fields are Wood functions with proper parameters defined through the `fs.test` function in the `fdaPDE` package in R. The covariates are generated as

$$X_{1,i} \sim \mathcal{N}(0, 1), \;\; X_{2,i} \sim \mathcal{E}(1) \;\; \forall i = 1, ..., n$$

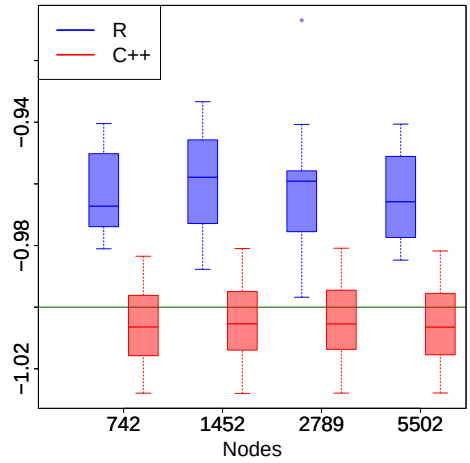
with true value $\boldsymbol{\beta} = (2, -1)^T$. For the representation of the data generation process we refer to A.0.2. The accuracy and the performance comparisons are shown in figure 4.24 and figure 4.25 for $\alpha = 50\%$.



(a) $RMSE$

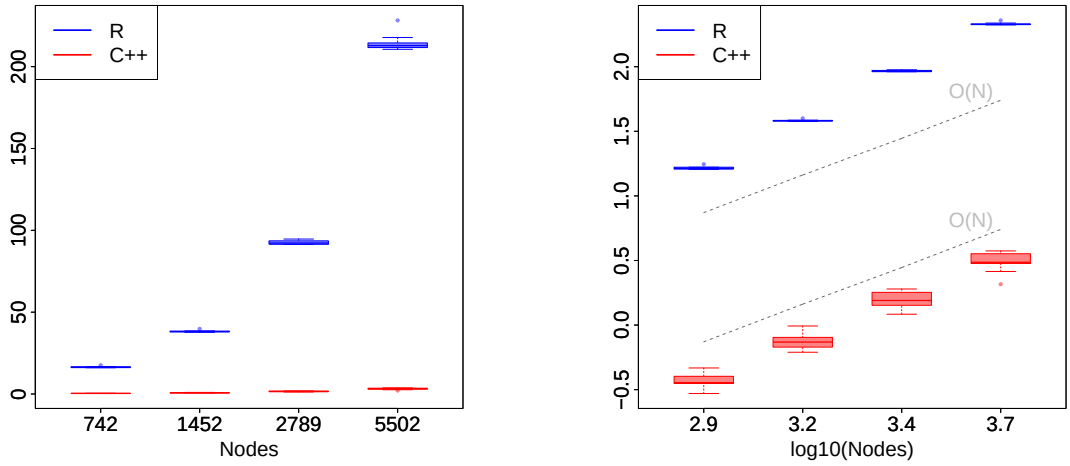


(b) Boxplot of $\hat{\beta}_1$ distribution around true value (darkgreen line)



(c) Boxplot of $\hat{\beta}_2$ distribution around true value (darkgreen line)

Figure 4.24: Test 9: Accuracy comparison between R and C++ over a 2D c-shaped domain with diverging mesh size. We can observe that the RMSE achieved by C++ is always lower, and less varying among the simulations, than the one obtained by R; anyway, for both software the values are good compared with the range of the standard deviation function which is [0.08, 0.93]. For the parametric part: the true value of $\boldsymbol{\beta}$ is always inside the C++ boxplot, instead R is biased for β_2 for all N. Therefore we can conclude that the C++ algorithm performs better in terms of overall accuracy.



(a) Computational time [seconds]

(b) Logarithmic scale in time [seconds]

Figure 4.25: Test 9: Performance comparison between R and C++ over a 2D c-shaped domain with diverging mesh size. Execution time is significantly lower in the C++ implementation and the discrepancy is much more evident for denser meshes. Moreover, we can see from the Log-Log plot on the right panel that the order in which time increases with respect to the number of nodes is linear in C++ and a bit more than linear in R.

4.3.2 Test 10: Semiparametric 2D unit square

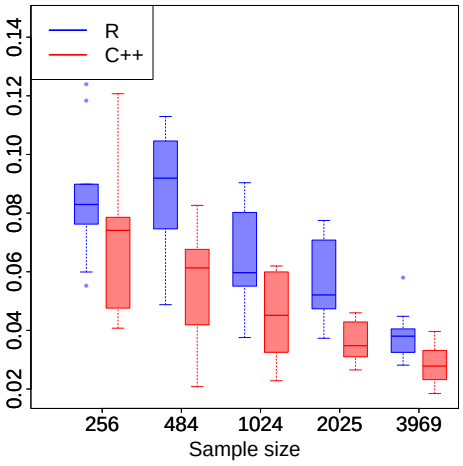
The second comparison test is run on a squared mesh with divergent sample size $n = [256, 484, 1024, 2025, 3969]$ and fixed number of nodes $N = 1936$. The setting is the same as Test 7.1 in 4.1. Specifically, data are generated according to strategy B in 4, where the deterministic field is

$$f(x, y) = 36\Big(\frac{1}{3}x^3 - \frac{1}{2}x^2\Big)\Big(\frac{1}{3}y^3 - \frac{1}{2}y^2\Big) \tag{4.10}$$

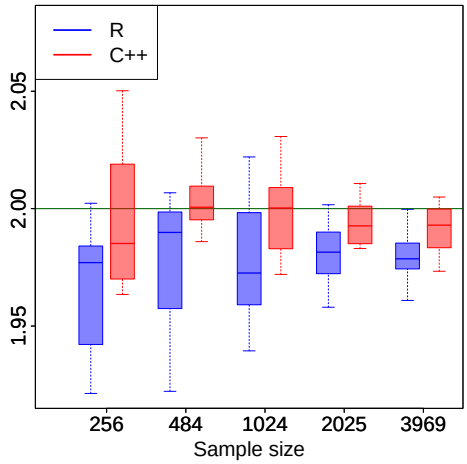
which satifies the Neumann boundary conditions. The covariates are generated as

$$X_{1,i} \sim \mathcal{N}(0, 1),\ \ X_{2,i} \sim \mathcal{E}(3)\ \ \forall i = 1, ..., n$$

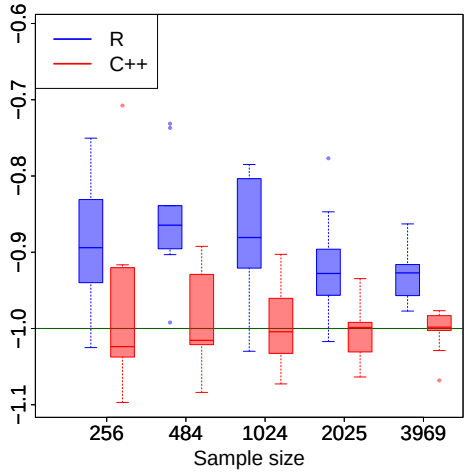
with true value $\boldsymbol{\beta} = (2, -1)^T$ and the noise vector is given by $\boldsymbol{\eta} \sim SN(\xi, \omega, \alpha_N)$. Choosing $\xi = 4$, $\alpha_N = 5$ and $\omega = 0.05\ range(f)$, the obtained standard deviation of the noise is around 0.5. The data geeration process is represented in A.0.3. The accuracy and the performance comparisons are shown in figure 4.26 and figure 4.27 for $\alpha = 50\%$.



(a) $RMSE$

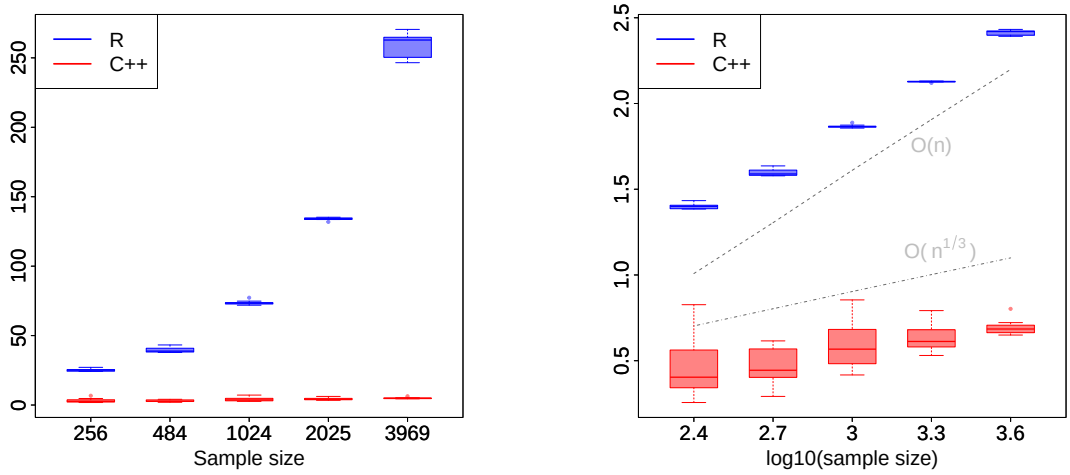


(b) Boxplot of $\hat{\beta}_1$ distribution around true value (darkgreen line)



(c) Boxplot of $\hat{\beta}_2$ distribution around true value (darkgreen line)

Figure 4.26: Test 10: Accuracy comparison between R and C++ over a 2D square domain with diverging sample size. The C++ solution has a statistically lower RMSE compared to the R one. In both the algorithms the RMSE decreases as the sample size increases, as expected. Moreover the two numerical solutions get closer , as n increases, due to the shrinking of the non-identificability region of the functional J (region where J is constantly equal to its minimum). For both $\hat{\beta}_1$ and $\hat{\beta}_2$, the C++ algorithm is closer to the true value than the R version. Therefore, as in Test 9, the C++ algorithm achieve better results in terms of overall accuracy.



(a) Computational time [seconds]

(b) Logarithmic scale in time [seconds]

Figure 4.27: Test 10: Performance comparison between R and C++ over a 2D square domain with diverging sample size. The C++ implementation is significantly faster than the R one and this is particularly evident as the sample size increases. From the Log-Log plot on the right panel we can see that the order in which time increases is sublinear in C++ (approximately $\frac{1}{3}$) and linear in R.

Chapter 5

Conclusions

To sum up, the main goal of our work has been the inclusion of a whole new class of statistical models SQR-PDE within the `fdaPDE` library.

The starting point has been the code developed by C.Castiglione, nevertheless our implementation differs from it in the resolution of semiparametric problems. Indeed, the versatility of the C++ language allowed us to move from a component-wise resolution, which usually leads to lower accuracy and slower convergence, to a simultaneous one thanks to the FPIRLS algorithm and the SR-PDE solver. From the performance comparison between R and C++ in 4.3, we can observe that our method results in comparable or even better accuracy outcomes and, above all, in a very significant speed up in time. Moreover, such speed up allowed us to consider the complete mass matrix $\mathbf{R}_0$ instead of its lumped version as in the R code, which guarantees a better accuracy as shown in 4.2.2.

For the definition of our new approach, we had to deal with two fundamental aspects. As first, we had to generalize the FPIRLS class in order to make it suitable to handle our problem; indeed, the FPIRLS class was initially designed to only manage GSR-PDE problems and so it showed some specific settings which have now been generalized also in view of future classes which could be added within the library. As second, we had to design our SQR-PDE class in agreement with the `fdaPDE` architecture in a such a way that we could exploit its methods and functionalities to achieve better performance and avoid code redundancy.

Finally, we proposed the Cholesky factorization as alternative method for the stochastic computation of the trace of the smoothing matrix $\mathbf{S}$. The latter method showed a better behaviour, compared with the computation based on the Woodbury decomposition, for relative small systems in terms of computational time. Thus, since our main aim is to

achieve the greatest possible efficiency, we chose to apply the Cholesky algorithm for problems with mesh size lower than a set threshold.

The main future direction of our work is the inclusion of spatio-temporal models in the framework of the spatial quantile regression, which is possible thanks to the new features of the `fdaPDE` library and that could allow to model several real applications of great interest, involving the evolution of phenomena with complicated spatio-temporal dependencies.

Chapter 6

Installation and instructions to run the tests

6.1 Installation

The source code described in this report can be downloaded as .zip file from https://github.com/DeSanctisDiBattista/PACS_project/tree/main that is a fork of the fdaPDE official repository. The following dependencies have to be installed on the system:

- a C++17 compliant compiler
- make
- CMake
- Eigen linear algebra library (version higher than 3.3)

6.2 Run GCV tests from C++

The GCV tests can be run directly from C++. The file `MainTest.cpp` includes all the tests that can be run (field, mesh, non-linear algebra, FEM and regression suites). Our tests are contained in `calibration/GCVTest.cpp`. To run the entire test suite locally, execute from shell

```
cmake CMakeLists.txt
make
./fdaPDE_test
```

Notice that the correspondent GCV test of Test 7 already implements the new mechanism of the automatic choice of the linear solver based on the problem's size. This means that the results contained in the report for those tests are not replicable from this test suite.

6.3 Run simulation tests from R

To run the simulation tests, we rely on a R/C++ wrapper. The structure for the wrapper class was already present in the library, hence we have extended it for the SQRPDE case in the `R_SQRPDE_ese.cpp` file. The latter defines the `R_SQRPDE` class and all the `RCPP_MODULE` to deal with the different settings of the implemented tests. Indeed, each `RCPP_MODULE` accounts for a choice of the dimension, the sampling strategy (including the areal case) and the penalization term. Finally, in order to deal with the linear network domain, we added a template specification for the class `RegularizingPDE` in the `Common.h` file for the case with $M = 1$ (local dimension of the mesh) and $N = 2$ (embeddding dimension of the mesh). Such specification is needed since in this case the neighbouring structure of the domain is stored in a sparse matrix, instead of a dense one.

Therefore the simulations tests can be run through the `Test_TestNumber.R` scripts present in the folder `tests_wrappers` of the repository. The numeration of the tests is the same one followed in the report. The only missing test is Test 7 which can be run directly in C++, as explained above, since it concerns only the GCV calculation.

In order to run the simulation tests the following packages have to be installed:

- `fdaPDE`: old version of the fdaPDE library
- `fdaPDE2`: new version of the fdaPDE library

The package `fdaPDE` is required to run the old R version of the code written by C.Castiglione and to have at disposal some functions to handle the mesh generation process. The package `fdapde2` is instead required to run our C++ code.

6.3.1 Installation of fdaPDE package in R

Run from the R terminal

```
1  install.packages('fdaPDE')
```

6.3.2 Installation of fdaPDE2 package in R

To install the package `fdaPDE2`, use `compile_fdaPDE2.R` in the repository. In particular, `Rcpp` and `RCppEigen` have to be installed first, if not done yet. The lines

```
system.file(package='Rcpp')
system.file(package='RcppEigen')
```

are meant to check whether such packages have been successfully installed (an empty string is returned if not).

Bibliography

[1] Eleonora Arnone et al. “Computationally efficient techniques for spatial regression with differential regularization”. In: (2023).

[2] Eleonora Arnone et al. “Modeling spatially dependent functional data via regression with differential regularization”. In: (2018).

[3] Cristian Castiglione. “Approximate inference for misspecified additive and mixed regression models”. PhD thesis. Università di Padova, 2023.

[4] Cristian Castiglione et al. “Spatial quantile regression with PDE regularization”. In: (2022).

[5] Aldo Clemente. *fdaPDE over Linear Networks*. PACS project. 2022.

[6] Alberto Colombo and Giulio Perin. *GSR-PDE*. PACS project. 2020.

[7] *fdaPDE*. Version 2. 2023. URL: https://github.com/fdaPDE/fdaPDE-core.

[8] J. Romero, A. Madrid, and J. Angulo. “Quantile-based spatiotemporal risk assessment of exceedances”. In: (2018).

[9] Laura M. Sangalli. “Spatial Regression With Partial Differential Equation Regularisation”. In: (2021).

[10] Laura M. Sangalli, James O. Ramsay, and Timothy O. Ramsay. “Spatial spline regression models”. In: (2013).

[11] K. M. Tan, L. Wang, and -X. Zhou W. “High-dimensional quantile regression: convolution smoothing and concave regularization”. In: (2022).

[12] L. Wang, Y. Wu, and R. Li. “Quantile regression for analyzing heterogeneity in ultra-high dimension”. In: (2012).

Appendix A

Appendix

A.0.1 Test GSRPDE

Here we present a simulation study for GSR-PDE models with Poisson areal data. This test is conducted in order to check both the changes made in the GSR-PDE class and the application of the FPIRLS algorithm with areal data, since the areal case was not tested before. Moreover, this is the first test which include the GCV calculation for GSR-PDE models in the new version of the `fdaPDE` library.

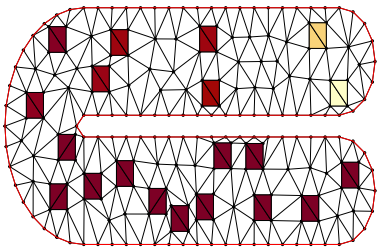
The simulation setting is the same of test Test 3, thus the data are collected over $n = 20$ subdomains in the C-shaped domain. The spatial field is again given by:

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

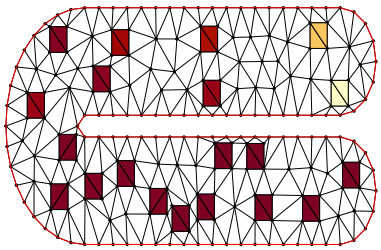
where $a(x, y)$ and $d(x, y)$ have the same definitions as the previous case. The data generation process now accounts for the employment of the Poisson distribution in the following way:

$$\begin{aligned} g(\mu(\mathbf{p})) = \theta(\mathbf{p}) = x_i\beta + \frac{1}{|D_i|}\int_{D_i} f \; d\mathbf{p}, \quad i = 1, ..., n \\ z_i \sim Poisson(\mu_i) \end{aligned} \tag{A.1}$$

where $\beta = 2$ and $\mathbf{x} \sim Beta(2, 2)$ is the covariate vector. The data together with the mean parameter $\boldsymbol{\mu}$ are shown in figure A.1. We consider a simple laplacian penalization and the smoothing parameter λ is chosen through the exact GCV computation. The simulation results are shown in figure A.2.

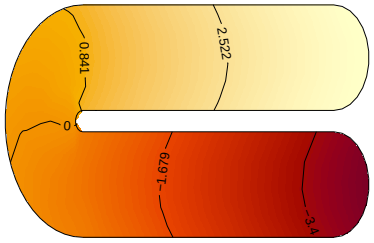


(a) Mean $\boldsymbol{\mu}$ over subdomains: high values yellow, low values red.

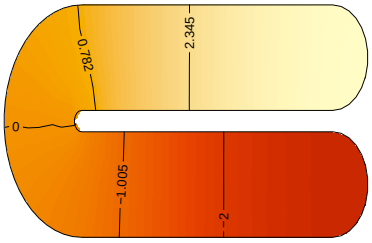


(b) Samples over subdomains: high values yellow, low values red.

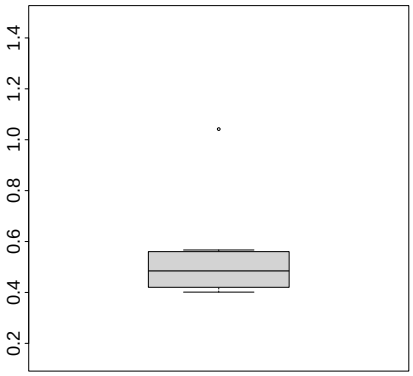
Figure A.1: Test GSR-PDE: Simulation setting for areal data. As we can notice, the values of the mean, and consequently of the sampled data, are concentrated over low values.



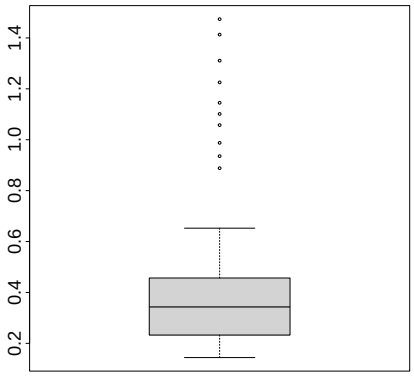
(a) True field



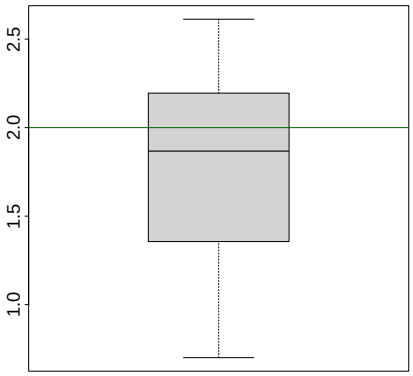
(b) Fitted field



(c) $RMSE_f$



(d) $SpRMSE_f$

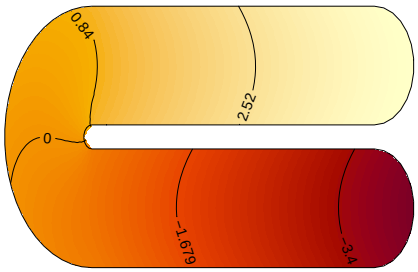


(e) Boxplot of $\hat{\beta}$ distribution around true value (darkgreen line)

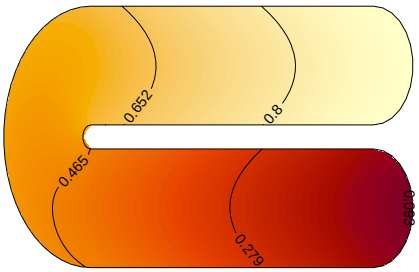
Figure A.2: Test GSR-PDE: Simulations results for areal data over a 2D C-shaped domain. The fitted field reconstructs well the true one, however we can notice that for lower values the discrepancy is more evident. RMSE and SpRMSE are good compared with the standard deviation of the true field $\sigma := \sqrt{\mu}$, which is in the range [0,3] for the larger part of the domain. Finally the true value of β is correctly inside the $\hat{\beta}$ boxplot.

A.0.2 Data setting for comparison tests on a C-shaped domain

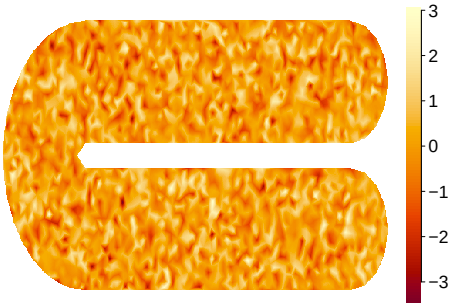
Here we show the data simulation settings for Test 7.2 and Test 9 over a C-shaped domain. Data are generated following strategy A in 4. In the figure below we show the mean and the standard deviation fields and the two covariates illustrated as their continuous interpolation.



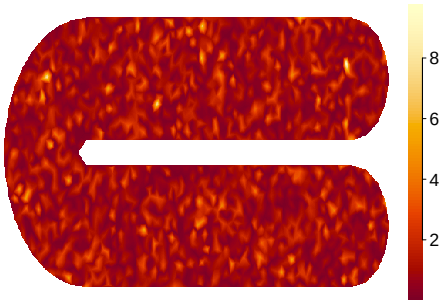
(a) Mean $\mu(x, y)$



(b) Standard deviation $\sigma(x, y)$



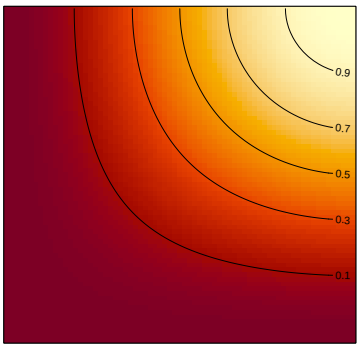
(c) First covariate $X_1 \sim N(0, 1)$



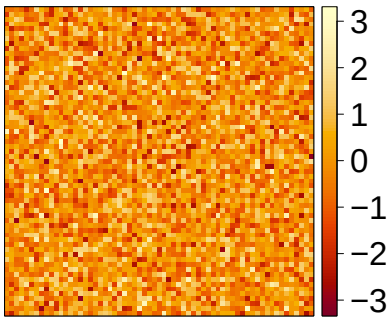
(d) Second covariate $X_2 \sim Exp(1)$

A.0.3 Data setting for comparison tests on a unit square domain

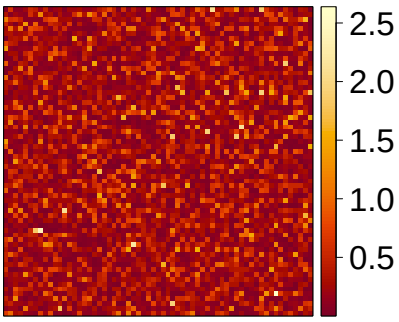
Here we show the data simulation settings for Test 7.1 and Test 10 over a unit square domain. Data are generated following strategy B in 4. In the figure below we illustrate the deterministic field and the two covariates shown as their continuous interpolation.



(a) Deterministic field $f(x, y)$



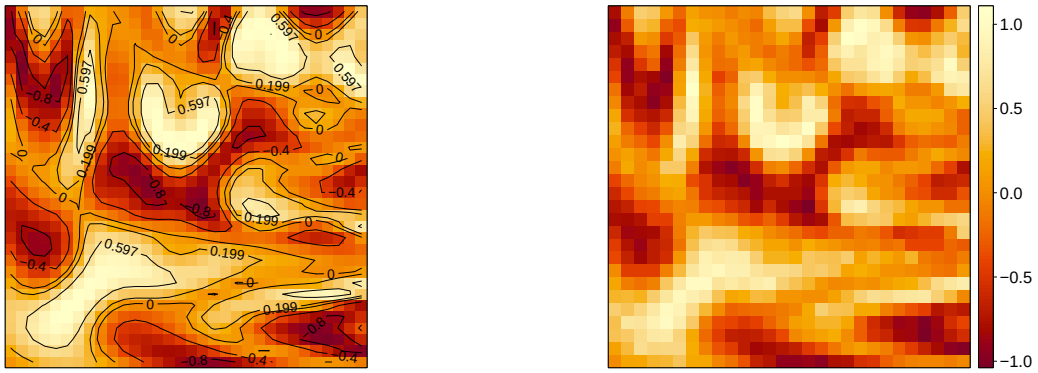
(b) First covariate $X_1 \sim N(0, 1)$



(c) Second covariate $X_2 \sim Exp(3)$

A.0.4 Data setting for Test 8

Here we show the data simulation settings for Test 8 meaning the test checking the effects of the mass lumping technique. The test is run over a square domain and the data are generated following strategy B in 4, without covariates. In the figure below we illustrate the deterministic field and the overall data represented as their continuous interpolation.



(a) Deterministic field $f(x, y)$

(b) Overall data