Regression Models
with Differential Regularization

Efficient computation of the Equivalent Degrees of Freedom

Clelia Bambini        Luca Giussani

April 3, 2017
# Contents

1 Mathematical model  
1.1 The framework  
1.2 Data and model  
1.3 Estimation problem  
1.4 Finite Element solution to the estimation problem  
1.5 Smoothing Matrix  
1.6 Previous work  

2 Solving the system  
2.1 Woodbury decomposition  
2.2 Implementation  
2.3 Conclusions  

3 GCV computation  
3.1 Two different approaches  
3.2 Implementation  
3.3 Conclusions  

4 Approximation of the GCV  
4.1 Stochastic GCV  
4.2 Implementation  
4.3 Conclusions  

5 SPLinearSolvers  
5.1 General framework  
5.2 Setting parameters  
5.3 Parallelization  
5.4 Example of application  

1
## 6 Tests

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Test coherence of the stochastic estimator</td>
<td>26</td>
</tr>
<tr>
<td>6.2 Test on variance</td>
<td>28</td>
</tr>
<tr>
<td>6.3 Test on scalability</td>
<td>29</td>
</tr>
<tr>
<td>6.4 Performance comparison: Exact VS Stochastic</td>
<td>30</td>
</tr>
</tbody>
</table>
Chapter 1

Mathematical model

1.1 The framework

The models we take into account lie between statistics and numerical analysis and have been developed in [2] and [1]. This class of models, that can go under the name of SR-PDE (Spatial Regression models with PDE penalization), aims at estimating surfaces and spatial fields when a priori knowledge on the phenomenon under study is available and can be formalized in terms of partial differential equation; this is particularly useful since many sciences use PDEs to describe complex phenomena. The modeling generalizations include the capacity to account for covariate measures, via semiparametric framework, and to comply with different conditions at the boundary of the domain.

A functional data analysis approach is used, as the problem consists of the minimization of a penalized least squares functional with a roughness penalty involving a partial differential operator. Then this problem is discretized and solved through the Finite Element Method. The advantages of this model can be seen from a statistical point of view: SR-PDE estimators are linear in the observed data values and so the classical inferential tools can be used. Also numerically, the Finite Element Method makes the problem computationally very efficient.

1.2 Data and model

Consider a bounded and regular domain $\Omega \subset \mathbb{R}^2$, whose boundary $\partial \Omega$ is a curve of class $C^2$. Let:

- $\{\mathbf{p}_i = (x_i, y_i); i = 1, \ldots, n\}$ be a set of $n$ points in $\Omega$;
• $z_i$ be the value of a real valued variable observed at point $p_i$, representing noisy evaluations of the field;

• $w_i = (w_{i1}, \ldots, w_{iq})^T$ be a vector of covariates associated to observation $z_i$.

The semi-parametric model is:

$$z_i = w_i^T \beta + f(p_i) + \epsilon_i \quad i = 1, \ldots, n$$  

(1.1)

where

• $\epsilon_i$ are errors independently distributed, with zero mean and variance $\sigma^2$;

• $\beta \in \mathbb{R}^q$ are the coefficients of the regression;

• $f : \Omega \to \mathbb{R}$ is real valued and differentiable and $\partial f / \partial n = 0$ on $\partial \Omega$.

Other boundary conditions can be considered, but for simplicity we limit the exposition to the homogeneous Neumann case.

1.3 Estimation problem

Let $J_\lambda(\beta, f)$ be the penalized sum-of-squares error functional:

$$J_\lambda(\beta, f) = \sum_{i=1}^n (z_i - (f(p_i) + \sum_{j=1}^q \beta_j w_{ij}))^2 + \lambda \int_\Omega |Lf - u|^2 d\Omega$$  

(1.2)

where $L$ is a linear second order elliptic operator, which can be written in a general form as

$$Lf = -\text{div}(K \nabla f) + b \cdot \nabla f + cf$$

where $K$ is the diffusion tensor, $b$ the transport vector and $c$ the reaction coefficient. Instead $\lambda$ is a parameter that needs to be finely tuned as it balances the trade off between the data fitting criterion (the least squares error) and the model fitting criterion, which penalizes the departures from the PDE describing the phenomenon. This functional is well defined if $f \in H^2(\Omega)$ thanks to the embedding $H^2(\Omega) \subset C(\overline{\Omega})$.

The regression coefficient vector $\beta$ and the function $f$ are estimated by minimizing the penalized sum-of-squares error functional:

$$\left( \hat{\beta}, \hat{f} \right) = \arg\min_{f \in H^2_{w0}} J_\lambda(\beta, f) \quad \beta \in \mathbb{R}^q$$  

(1.3)

2
where $H_{n0}^2 = \{ f \in H^2 : \frac{\partial f}{\partial n} = 0 \}$.

Denote by $W = [w_{ij}] \in \mathbb{R}^{n \times q}$ the matrix whose i-th row holds the q covariates associated to $z_i$. Then $P$ is the projector on the subspace generated by the columns of $W$, therefore

$$P = W(W^TW)^{-1}W^T \quad (1.4)$$

Then let $Q = I - P$, where $I$ is the identity matrix.

The minimization problem (1.3), under suitable regularity conditions for $L$ and boundary conditions on $f$, has a unique solution. (For the proof see [3]).

**Proposition 1.** The estimators $\hat{f}$ and $\hat{\beta}$ that jointly minimize (1.2) over $\beta \in \mathbb{R}^q$ and $f \in H_{n0}^2$ are uniquely determined and are:

- $\hat{\beta} = (W^TW)^{-1}W^T(z - \hat{f}_n)$
- $\hat{f}$ satisfying

$$v_n^TQ\hat{f} + \lambda \int_\Omega (Lv)(L\hat{f}) = v_n^TQz + \int_\Omega u(Lv) \quad \forall v \in H_{n0}^2(\Omega) \quad (1.5)$$

where $\hat{f}_n = (\hat{f}(p_1), \ldots, \hat{f}(p_n))^T$ and $v_n = (v(p_1), \ldots, v(p_n))^T$ are the vectors of the evaluations of $\hat{f}$ and $v$ at the $n$ data locations.

### 1.4 Finite Element solution to the estimation problem

The estimation problem is infinite dimensional and it can’t be solved analytically. PDEs are usually solved in weak sense and then the weak problem is discretized by means of the Finite Element Method. In order to define the weak problem associated to (1.5) we introduce the bilinear form $a(\cdot, \cdot)$ associated to the operator $L$, defined as

$$a(f, u) = \int_\Omega [K\nabla f \cdot \nabla u + (b \cdot \nabla f)u + cfu] \quad (1.6)$$

Then problem (1.5) may be reformulated as follows:

find $(\hat{f}, \hat{g}) \in V = (H_{n0}^1(\Omega) \cap C^0(\Omega)) \times H^1(\Omega)$ that satisfy

$$\begin{cases} v_n^TQ\hat{f} + \lambda a(\hat{f}, v_1) = v_n^TQz \\ - \int_\Omega \hat{g}v_2 + a(\hat{f}, v_2) = \int_\Omega uv_2 \end{cases} \quad (1.7)$$
for each $v_1, v_2 \in V$, where $v_1 = (v_1(p_1), \ldots, v_1(p_n))^T$.

Let’s consider a regular triangulation $T$ of the domain, where adjacent triangles share either a vertex or an edge. The triangulation points in $T$ may or may not coincide with the data locations $p_i$.

Let $\xi_k$ with $k = 1, \ldots, K$ be the nodes on which the basis functions are defined. The order of the polynomials we choose influences the number of nodes: in the linear case we need three basis functions over each triangle, in the quadratic case we need six basis functions and therefore six nodes over each triangle. A piecewise linear/quadratic basis function $\psi_k$ is associated to each $\xi_k$ and it has value 1 at the $k$-th node and value 0 at all the other nodes. The basis functions built this way are called Lagrange Finite Elements and they define a subspace of $H^1(\Omega)$, which we call $H^1_T$.

Setting $\psi = (\psi_1, \ldots, \psi_K)^T$ and $f = (f(\xi_1), \ldots, f(\xi_K))^T$ we can define any $f \in H^1_T$ by its value at the $K$ nodes:

$$f(x, y) = \sum_{k=1}^K f(\xi_k) \psi_k(x, y) = f^T \psi(x, y) \quad (1.8)$$

The Finite Element Method is obtained by taking $V = H^1_T$ in (1.7); expressing the functions $v_1, v_2, \hat{f}, \hat{g}$ in the form (1.8) leads to a linear system of equations.

In order to write the algebraic form of the problem, let’s introduce the order $K$ matrices $R_0$ and $R_1$, defined by:

$$[R_0]_{ij} = \int_{\Omega} \psi_j \psi_i d\Omega$$

$$[R_1]_{ij} = \int_{\Omega} (K \nabla \psi_j \cdot \nabla \psi_i + (b \cdot \nabla \psi_j) \psi_i + c \psi_j \psi_i) d\Omega$$

and the matrix $\Psi \in \mathbb{R}^{n \times K}$, given by the evaluation of the basis functions at the $n$ data locations:

$$\Psi = \begin{bmatrix} \psi^T(p_1) \\ \vdots \\ \psi^T(p_n) \end{bmatrix}$$

**Proposition 2.** The estimators $\hat{\beta} \in \mathbb{R}^q$ and $\hat{f} \in H^1_T$ that solve the discrete counterpart of the estimation problem (1.7) exist, are unique and are given by:

- $\hat{\beta} = (W^T W)^{-1} W^T (z - \hat{f}_n)$
• \( \hat{f} \) is identified by the coefficient vector \( \hat{f} \) satisfying

\[
\begin{bmatrix}
\Psi^T Q \Psi & -\lambda R_1 \\
-\lambda R_1^T & -\lambda R_0
\end{bmatrix}
\begin{bmatrix}
\hat{f} \\
\hat{g}
\end{bmatrix}
= \begin{bmatrix}
\Psi^T Q z \\
u
\end{bmatrix}
\tag{1.9}
\]

From here on we define

\[
M = \begin{bmatrix}
\Psi^T Q \Psi & -\lambda R_1 \\
-\lambda R_1^T & -\lambda R_0
\end{bmatrix}
\tag{1.10}
\]

1.5 Smoothing Matrix

Once \( \hat{f} \) and \( \hat{\beta} \) are estimated through the Finite Element Method, the fitted values \( \hat{z} \) can be computed as follows:

\[
\hat{z} = W \hat{\beta} + \Psi \hat{f} = Sz + r
\]

where the smoothing matrix \( S \) and the vector \( r \) are defined as

\[
S = P + Q \Psi M^{-1}(1 : K, 1 : K) \Psi^T Q 
\tag{1.11}
\]

\[
r = -Q \Psi M^{-1}(1 : K, 1 : K) \lambda R_1 R_0^{-1} u 
\tag{1.12}
\]

Thanks to the linearity of the estimator \( \hat{z} \) in the observation we can use \( tr(S) \) as a measure of equivalent degrees of freedom for linear estimators. By the cyclic property of the trace and the idempotence of \( Q \) we have:

\[
tr(S) = tr(P) + tr(Q \Psi M^{-1}(1 : K, 1 : K) \Psi^T Q) \\
= q + tr(\Psi M^{-1}(1 : K, 1 : K) \Psi^T Q) 
\tag{1.13}
\]

The equivalent degrees of freedom are useful in order to choose the smoothing parameter \( \lambda \). Indeed a possible criterion to select \( \lambda \) is the minimization of the Generalized Cross Validation index

\[
GCV(\lambda) = \frac{1}{n(1 - tr(S/n))} (\hat{z} - z)^T (\hat{z} - z) 
\tag{1.14}
\]

1.6 Previous work

This model has been implemented in \( \texttt{R} \) and also in \( \texttt{C++} \) (with an \( \texttt{R} \) interface) inside the package fdaPDE [8]. In this context our work focuses on the calculation of the GCV, because the initial implementation, which closely follows formula (1.13), involves the explicit inversion of \( M \). This naive approach is
highly inefficient because the dimensions of matrix $M$ are large and moreover the matrix is generally dense.

The first step to improve the computation is to investigate on the structure of matrix $M$. In Chapter 2 we show how, thanks to Woodbury matrix decomposition, we can develop an efficient procedure to solve linear systems involving matrix $M$. In Chapter 3 we work on the formula of the equivalent degrees of freedom in order to improve the performances using specific libraries. Since these approaches are not yet completely successful, Chapter 4 analyzes a novel approach for the estimation of the GCV which involves a stochastic estimator of the trace of matrix $S$. Chapter 5 presents an interface we developed to abstract the resolution of a linear system from the library chosen to solve it. The last Chapter describes some tests on the performance of our procedure.
Chapter 2

Solving the system

2.1 Woodbury decomposition

In multiple points of the code there is the need to solve a linear system involving matrix (1.10)

\[ M = \begin{bmatrix} \Psi^T Q \Psi & -\lambda R_1 \\ -\lambda R_1^T & -\lambda R_0 \end{bmatrix} \] (2.1)

which belongs to \( \mathbb{R}^{2K \times 2K} \), where \( K \) is the number of nodes. The peculiarity of this matrix, especially when covariates are considered, is that the north-west block is not sparse but it has a special structure. More precisely, the matrix \( Q \) is full, but can be expressed as \( Q = I - P \) where \( P \) is the matrix (1.4) whose rank is equal to the number of covariates of the model. Therefore we can write \( M \) as the sum of a sparse matrix \( A \) and a low rank matrix \( B \).

\[ M = \begin{bmatrix} \Psi^T \Psi & -\lambda R_1 \\ -\lambda R_1^T & -\lambda R_0 \end{bmatrix} + \begin{bmatrix} \Psi^T (-P) \Psi & 0 \\ 0 & 0 \end{bmatrix} = A + B \] (2.2)

In this setting there is a result about the inverse of the sum of two matrices, one of them being low rank:

**Proposition 3** (Woodbury Matrix Identity).

\[ M^{-1} = (A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + V A^{-1} U)^{-1} V A^{-1} \] (2.3)

where \( A \in \mathbb{R}^{l \times l} \), \( U \in \mathbb{R}^{l \times m} \), \( C \in \mathbb{R}^{m \times m} \), \( V \in \mathbb{R}^{m \times l} \).

In order to apply this proposition to matrix \( M \) we only need to express \( B \) as product of \( UCV \). Using (1.4) we can see that the north-west block of
\( B \) can be written as
\[
\Psi^T(-P)\Psi = \left( \Psi^TW \right) \left( -W^TW \right)^{-1} (W^T\Psi) = \tilde{U}\tilde{C}\tilde{V}
\]

Therefore defining
\[
U = \begin{bmatrix} \Psi^TW \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{U} \\ 0 \end{bmatrix} \in \mathbb{R}^{2K \times q}
\]
\[
C = -(W^TW)^{-1} = \hat{C} \in \mathbb{R}^{q \times q}
\]
\[
V = [ \ W^T\Psi \ | \ 0 \ ] = [ \ \tilde{V} \ | \ 0 \ ] \in \mathbb{R}^{q \times 2K}
\]
we can write \( B = UCV \).

We have hence derived that \( M = (A + UCV) \), thus allowing the usage of the Woodbury matrix identity for the factorization of the matrix \( M \). There are two different advantages to this approach:

- There is no need to factorize the dense matrix \( M \). Instead we only need to factorize the sparse matrix \( A \) and the matrix \( (C^{-1} + VA^{-1}U) \), which is dense but very small (\( \in \mathbb{R}^{q \times q} \), \( q \) being the number of covariates considered in the model).

- Reduced memory consumption: instead of storing the matrix \( M \) which has a dense north-west block, we store the sparse matrix \( A \) and the factors \( \Psi \) and \( W \).

### 2.2 Implementation

We introduced this procedure in the previously existing code by adding three methods to the class \texttt{MixedFERegression}: \texttt{system\_factorize}, \texttt{system\_solve} and \texttt{LeftMultiplyByQ}.

\textbf{System\_factorize}  This method takes care of the factorization of the sparse matrix \( A \). When the problem takes into account covariate measures then it's necessary to compute also the factorization of \( (C^{-1} + VA^{-1}U) \) that we call \( G \).
System _solve_ This method implements the resolution of a linear system taking advantage of the Woodbury formula

\[ M^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1} \]

thus reducing the problem to a chain of linear systems. Therefore solving \( Mx = b \) amounts to:

1. Solve the system
   \[ Ax_1 = b \]
2. Set
   \[ x_2 = Vx_1 \]
3. Solve the system
   \[ Gx_3 = x_2 \]
4. Set
   \[ x_4 = Ux_3 \]
5. Solve
   \[ Ax_5 = x_4 \]
6. And finally, set
   \[ x = x_1 - x_5 \]

In order to solve the system, besides the step 3, whose cost is negligible, we are solving twice a system whose matrix is \( A \). However, it is computationally more convenient to proceed this way, thanks to the sparsity of matrix \( A \) as it was outlined in the previous section.

LeftMultiplyByQ Since the problem of multiplying matrix \( Q \) times a generic other vector \( u \) is frequent, we have developed this method. The main issue is that \( Q \) is a dense matrix but, exploiting its decomposition, we defined a procedure that doesn’t require the storage of the matrix. Therefore the product becomes:

\[ Qu = (I - P)u = u - W(W^T W)^{-1}W^T u \]  

and this can be solved in some steps:

1. If covariates are present, factorize
   \[ W^T W \]
2. Solve the system

\[(W^T W)x_1 = W^T u\]

3. Finally

\[Qu = u - Wx_1\]

With this procedure there is no need to store the whole matrix \(Q\) and this helps in terms of memory occupation.

2.3 Conclusions

With this approach we reached two different goals. It is now possible to solve the linear systems without storing all the coefficients, therefore reducing memory consumption. More importantly, there is no need anymore to factorize the dense matrix \(M\) as it is sufficient to factorize the sparse matrix \(A\). This approach makes possible to solve conspicuously bigger problems with respect to a naive approach.
Chapter 3

GCV computation

3.1 Two different approaches

The Generalized Cross Validation has a great importance in penalized regression models as it is able to suggest the choice of the best parameter $\lambda$. As it was mentioned in section 1.5, we want to select the $\lambda$ that minimizes

$$GCV(\lambda) = \frac{1}{n(1 - \text{tr}(S/n))^2} (\hat{z} - z)^T (\hat{z} - z)$$  \hspace{1cm} (3.1)

This index requires the calculation of $\text{tr}(S)$, the trace of the smoothing matrix $S$, which is a measure of the equivalent degrees of freedom for linear estimators. The calculation can be done with two different but equivalent procedures. In the introduction to the mathematical model the following formula was presented:

$$\text{edf} = \text{tr}(S) = q + \text{tr}(\Psi M^{-1}[1 : K, 1 : K] \Psi^T Q)$$  \hspace{1cm} (3.2)

but it can be useful to write the inverse of $M$ in another form. Indeed it is true that:

$$M^{-1}[1 : K, 1 : K] = (\Psi^T Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1}$$

Therefore the equivalent degrees of freedom can be computed also like this

$$\text{edf} = \text{tr}(S) = q + \text{tr}(\Psi(\Psi^T Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} \Psi^T Q)$$  \hspace{1cm} (3.3)

The two formulas can always be used, but we can divide the possible problems into four different categories in order to choose the more suitable formula for each case. The variability is given by the fact that covariates can
be present and not, and that the nodes of the mesh can and cannot be in the same locations of the observations.

We note the following two facts:

- When covariates are not considered in the model, $P$ (the projector on the subspace generated by the covariates) is the null matrix. Hence $Q$ is the identity matrix, because $Q = I - P$.

- When all the locations $p_i$ of the observations are situated on nodes $\xi_k$ of the computational grid, the matrix $\Psi$ assumes a particular structure. More precisely:
  - it is composed of 0s and 1s;
  - the columns whose index corresponds to a node where there are no observations are empty;
  - there is a single element of value 1 in the columns whose index corresponds to a node where there is an observation. Its row index coincides with the index of that observation.

We denote a matrix $\Psi$ possessing such a structure with the symbol $\hat{\Psi}$.

These observations thus allow to distinguish four different cases:

**Case 1**: $\Psi = \hat{\Psi}$ and $Q = I$

**Case 2**: $\Psi \neq \hat{\Psi}$ and $Q = I$

**Case 3**: $\Psi = \hat{\Psi}$ and $Q \neq I$

**Case 4**: $\Psi \neq \hat{\Psi}$ and $Q \neq I$

In each one of these cases we selected the approach which gives the best results for the computation of the equivalent degrees of freedom.

**Case 1** Case 1 is the most advantageous from the computational point of view: in fact formula (3.2) reduces to

$$\text{edf} = q + \sum_{i=1}^{n} M^{-1}[k_i, k_i]$$

where $k_i$ is the index of the node assigned to the $i$-th observation. This form is useful because it requires the computation of only $n$ entries of $M^{-1}$, and not $K^2$ as formula (3.2). This observation permits to
employ a functionality of the library MUMPS which enables an efficient computation of selected entries of the inverse of a matrix. Hence in this case we can avoid an explicit inversion of matrix $M$. For the implementation details we refer to the next section.

**Case 2, 3 and 4** For the other cases, the aforementioned approach is no longer convenient, as the presence of the matrices $Q$ or $Ψ$, which multiply the inverse matrix, would require the calculation of an increasing number of entries of $M^{-1}$. This would not be ideal as the MUMPS routine is efficient only when the number of entries to be computed are $O(K)$ and not $O(K^2)$ as it would become in these cases. Since this approach is no longer viable, we need to resort to an explicit computation of the inverse of the matrix. We ascertained that formula (3.3) is more efficient:

$$edf = q + tr(Ψ(Ψ^TQΨ + λR_1^TR_0^{-1}R_1)^{-1}Ψ^TQ)$$

### 3.2 Implementation

The existing R implementation used the second formula (3.3) to solve the problem, whereas the C++ implementation always followed the second one (3.2). In any case the time required by these calculations, which need to be called several times as $λ$ varies, is too time consuming to be actually used. We need not to forget that the GCV is calculated only to choose the correct value of $λ$ and it is not required to solve the system. The computation of the edf is done inside the method `MixedFERegression::ComputeDegreesOfFreedomExact`. This method recognizes the case of the problem at hand and performs the computation accordingly.

**Case 1** This case is handled with MUMPS, “MUltifrontal Massively Parallel Solver” (see e.g. [6] and [7]), which is a library for solving systems of linear equations of the form $Ax = b$ and it is particularly suitable because it gives the possibility to compute efficiently selected entries of the inverse of matrix $A$.

In order to be able to use this particular feature of MUMPS, after having initialized the structure, we need to set the parameter’s array ICNTL; $ICNTL(30) = 1$ to compute the elements of the inverse corresponding to non-zero elements of the right hand side and $ICNTL(20) = 1$ to specify that the right hand side has to be sparse. The information about the matrix $M$ need to be communicated through the following
arrays:
- \texttt{nz\_rhs} number of entries of the inverse matrix to be computed
- \texttt{nrhs} number of columns of the inverse matrix
- \texttt{irhs\_ptr} pointers to the columns
- \texttt{irhs\_sparse} array of row indexes

The solution will be stored in the array \texttt{rhs\_sparse} which needs to be allocated beforehand. We remark that at this stage we used only the serial version of \textsc{MUMPS}, even though the calls to \textsc{MUMPS} always require \texttt{MPI}.

Case 2, 3 and 4 Computing the equivalent degrees of freedom, using the cyclic property of the trace, becomes

\[ q + \text{tr}( (\Psi^T Q \Psi + \lambda R_1^T R_0^{-1} R_1)^{-1} \Psi^T Q \Psi ) \]

and this is carried out following the chain of steps:

1. With the process explained in (2.4), build the matrix

\[ X_1 = \Psi^T Q \Psi \]

2. Compute the LU factorization of \( R_0 \)

3. Solve the \( K \) linear systems

\[ R_0 X_2 = R_1 \]

4. Build the matrix

\[ X_3 = X_1 + \lambda R_1^T X_2 \]

5. Compute the Cholesky factorization of \( X_3 \) using \texttt{Eigen}

6. Solve the \( K \) linear systems

\[ X_3 X = X_1 \]

7. Compute the trace of \( X \)

In the cases 2 and 3, the fact that, respectively, \( Q = I \) or \( \Psi = \hat{\Psi} \) is taken into account to assemble in an efficient way the matrices involved in the computation.
3.3 Conclusions

The choice of using ad hoc formulas and strategies for each particular case brings an important improvement in terms of computing time, especially in Case 1. Table 3.1 compares the performances of fdaPDE (original code) and fdaPDE2 (our version) in a test in which the locations of the observations coincide with the nodes of the triangulation ($\Psi = \hat{\Psi}$) and covariates are not included ($Q = I$). As the number $K$ of nodes increases from $K = 400$ to $K = 4900$ we can see a clear improvement of the performance.

<table>
<thead>
<tr>
<th>$K$</th>
<th>Time [s]</th>
<th>fdaPDE</th>
<th>fdaPDE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>0.31</td>
<td>0.07</td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>3.11</td>
<td>0.67</td>
<td></td>
</tr>
<tr>
<td>1600</td>
<td>17.66</td>
<td>3.97</td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>95.37</td>
<td>13.44</td>
<td></td>
</tr>
<tr>
<td>3600</td>
<td>342.57</td>
<td>49.95</td>
<td></td>
</tr>
<tr>
<td>4900</td>
<td>killed</td>
<td>163.8</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Time required by function smooth.FEM.basis

Moreover, when the number of nodes in the mesh is $K = 4900$, fdaPDE saturates the memory of the machine used to perform the test and the computation cannot be completed, whereas fdaPDE2 manages to complete the computation. Nonetheless, when the size of the problem grows, that is when the number of nodes $K$ becomes large, the computational time becomes too high also with this approach. This is especially true if we consider that the calculation of the edf needs to be repeated to fine tune the parameter $\lambda$, therefore this obstacle makes the calculation impractical.
Chapter 4

Approximation of the GCV

As the first results on the GCV weren't completely successful and the time required by the calculation presented in the previous chapter couldn't decrease consistently, we changed technique. In this chapter we present a different way to compute the GCV, starting from the work of Hutchinson in [4].

4.1 Stochastic GCV

We start by presenting the following result:

**Lemma 1.** Let $S$ be a symmetric matrix and let $\mathbf{u} = (u_1, \ldots, u_N)$ be a vector of $N$ independent samples from a random variable $U$ which takes values $-1$ and $1$ each with probability $1/2$. Then

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

and

$$Var[\mathbf{u}^T S \mathbf{u}] = 2 \sum_{i \neq j} ([S]_{ij})^2$$

We want to calculate an approximation of the Generalized Cross Validation using this stochastic unbiased estimator of the trace of a matrix. In this chapter we consider the formula (3.2) to compute the GCV, that is

$$edf = tr(S) = q + tr(\Psi M^{-1}[1 : K, 1 : K]\Psi^T Q)$$

Therefore, applying this estimator, the computation of the edf becomes

$$edf = tr(S) = q + E[\mathbf{u}^T \Psi M^{-1}[1 : K, 1 : K]\Psi^T Q \mathbf{u}]$$
We can see $M^{-1}[1 : K, 1 : K]$ as a product of suitable block-wise identity matrix times the whole matrix $M$. With

$$I_K = \begin{bmatrix} I \\ 0 \end{bmatrix} \in \mathbb{R}^{2K \times K}$$

then

$$edf = tr(S) = q + \mathbb{E}\left[u^T \Psi I_K^T M^{-1} I_K \Psi^T Qu\right]$$

This estimator could operatively be used to calculate the edf, as its calculation requires the solution of one linear system of order $2K$, instead of $K$ linear systems as it was needed with the exact approach in the previous chapter. On the other hand the estimator is stochastic and we are therefore losing in precision of the result. To lower the variance of the estimator it is possible to repeat the procedure with any desired number of samples form the random variable U and then to average the results. In our experiments we found that acceptable accuracy can be achieved with a number of random vectors much smaller than $K$, thus greatly decreasing the computational requirements of the first approach.

### 4.2 Implementation

The R interface has been upgraded in order to give to the user the possibility to choose which method is to be used to compute the GCV. The user may also choose the number of realizations to be used ($nreal$). Inside the C++ code instead, method MixedFERegression::computeDegreesOfFreedom switches to computeDegreesOfFreedomExact or computeDegreesOfFreedomStochastic according to the value defined by the user.

Method computeDegreesOfFreedomStochastic implements formula (4.1):

$$edf = tr(S) = q + \mathbb{E}\left[u^T \Psi I_K^T M^{-1} I_K \Psi^T Qu\right]$$

It should be noted that in the code, for efficiency reasons, instead of replicating the computation of formula (4.1) $nreal$ times, a single realization is
made, using a stochastic matrix $U \in \mathbb{R}^{n \times \text{nreal}}$ instead of the stochastic vector $u \in \mathbb{R}^{p}$.

The computation is organized in the following steps:

1. Use a standard default random engine and a standard Bernoulli distribution to populate the matrix $U \in \mathbb{R}^{n \times \text{nreal}}$ with values 1 and -1 with equal probability.

2. With the procedure explained in (2.4), build the matrix $X_1 = I_K \Psi^T QU$

$$X_1 = \left[ \begin{array}{c} \Psi^T QU \\ 0 \end{array} \right] \in \mathbb{R}^{2K \times \text{nreal}}$$

3. At this point matrix $M$ is already factorized (from the previous solution of the FEM system); call the method `system_solve` to solve the \text{nreal} linear systems

$$MX_2 = X_1$$

4. Build the matrix

$$X_3 = U^T \Psi$$

5. The $i$-th estimate $\hat{edf}_i$ is given by

$$\hat{edf}_i = q + X_3[i, :] \cdot X_2[1 : K, i]$$

for $i = 1, \ldots, \text{nreal}$

6. Compute the sample mean of the values $\hat{edf}_i$:

$$\hat{edf} = \frac{1}{\text{nreal}} \sum_{i=1}^{\text{nreal}} \hat{edf}_i$$

Of course, if the model includes covariates, the system at step 3 is solved using the procedure based on Woodbury formula described in Chapter 2.

### 4.3 Conclusions

We have developed a procedure that, combining a stochastic estimator and a matrix decomposition approach, allows a much more efficient estimation of the equivalent degrees of freedom compared to a naive approach. In particular the use of the stochastic estimator brings the number of linear
system to be solved from $K$ (number of nodes of the grid) to a number $n_{real}$ which can be chosen much smaller than $K$, still maintaining a good accuracy. Moreover the Woodbury formula presented in Chapter 2 makes the solution of these systems very efficient even when covariates (which make $M$ dense) are included in the model.

Using the stochastic approach it is possible to tackle problems of a much bigger size with respect to the exact approach: in Chapter 3 we showed that the existing code struggled with 4900 nodes considering both the memory occupation and the computational time. In Chapter 6 we will see that instead the stochastic approach can easily handle problems with tens of thousands of nodes.
Chapter 5

SPLinearSolvers

As it can be seen in the previous chapters, in multiple points of the code there is the need to solve linear systems of equations. Today several linear algebra libraries are available that allow the efficient solution of linear systems. Each one of them has its relative strengths and weaknesses, whose assessment is most easily done experimentally, by applying the library to the problem at hand and evaluating its performance. One of the difficulties of this approach resides in the fact that each one of these libraries has its own interface. If the interface of a particular library is directly used when the solution of a linear system is required, then if the need arises to later change the choice of library, an extensive and non-local modification of the code would be required. Indeed the code should be rewritten wherever the previous library was used.

In order to avoid this problem, it would be preferable to decouple the interface used to manage a linear system from the library actually used to solve it. This can be achieved by defining an abstract interface which acts as a wrapper around the interfaces of the various libraries considered. This allows to switch between the different linear solvers without practically any change in the code which uses the abstract interface.

To suit the needs of our application, we developed an interface for the direct solution of sparse linear systems through matrix factorization.

5.1 General framework

The interface is provided by the abstract base class SpLinearSolver, whose definition is

```cpp
class SpLinearSolver {
```
protected:
    Eigen::MatrixXd _sol;
public:
    virtual ~SpLinearSolver() = default;
    virtual void factorize
        (const Eigen::SparseMatrix<double> &)=0;
    virtual void solve
        (const Eigen::MatrixXd &)=0;
    virtual const Eigen::MatrixXd &getSolution()
        { return _sol; }
    virtual void setParameters
        (const ParameterList &list) {};
};

As it can be seen, the interface uses Eigen data types as input and output parameters. This was done because Eigen data types are used throughout the code and represent an efficient and clean way to manipulate matrices and vectors in C++.

For the needs of our application we implemented the interfaces to two different solvers: EigenSparseLU for the LU factorization offered by the Eigen library and MumpsSparse for the parallel solver MUMPS. In particular we use this interface to factorize the matrix $A$ in (2.2). The user of the R interface can choose between the two solvers and the instantiation of the appropriate solver is done at runtime through a factory. To add a new factorization method to this framework, it is necessary to create a subclass of SpLinearSolver which must define the virtual methods factorize, solve and possibly setParameters. While the methods factorize and solve are merely wrappers around the corresponding library functions, setParameters requires some explanation.

5.2 Setting parameters

Many libraries for the direct solution of linear systems, besides the basic functionality (namely factorizing a matrix and solving a linear system), offer the possibility to set some parameters. In general these parameters vary between different libraries. Hence someone who wants to write a common interface to these libraries is faced with the problem of how to handle these parameters. Several approaches are possible:

- the parameters could be set to reasonable default values without giving possibility to the end user to alter them. This strategy is the simplest from the programming point of view, but it clearly limits the usage of
the interface to those problems for which the default values chosen for the parameters are adequate;

• for each parameter offered by the library, a corresponding method to set it could be added in the concrete subclass of SpLinearSolver. While providing access to all of the functionality of the library, this approach undermines the same concept of having a common interface between the various libraries.

Instead we used a simplified version of the approach used in the library Amesos.

The parameters that are to be passed to the solver are placed in a ParameterList, which contains a map that associates a parameter name to its value:

```cpp
class ParameterList {
    private:
        std::unordered_map<std::string,
                        std::unique_ptr<AbstractParameter>> _map;
    public:
        ParameterList() = default;
        template <typename T>
        void set(const std::string &s, T value);
        template <typename T>
        T getValue(
            const std::string &name,
            T defaultValue
        ) const;
};
```

The map does not store directly the values of the parameters, but rather a pointer to an object of type AbstractParameter, which is basically an empty class:

```cpp
class AbstractParameter {
    public:
        virtual ~AbstractParameter() = default;
};
```

This additional level of indirection allows the maximum flexibility in the types of parameters that can be passed to the solver. In fact AbstractParameter is subclassed by the template class Parameter:

```cpp
template <typename T>
```
```cpp
class Parameter: public AbstractParameter {
    private:
    T _value;
    public:
    Parameter(const T &value): _value(value) {};
    void set(const T &value) {_value = value;}
    T getValue() const {return _value;}
};
```

which provides the definition for a family of classes that can hold a generic object and provide methods to set and get its value. The definition of the setter and getter methods are provided below:

```cpp
template <typename T>
void ParameterList::set(const std::string &name, T value) {
    std::unique_ptr<Parameter<T>>
    value_ptr (new Parameter<T>(value));
    _map[name] = std::move(value_ptr);
}
```

```cpp
template <typename T>
T ParameterList::getValue(const std::string &name, T defaultValue) const {
    auto iter = _map.find(name);
    if (iter != _map.end()) {
        auto abstract_ptr = iter->second.get();
        auto concrete_ptr =
            dynamic_cast<Parameter<T>*>(abstract_ptr);
        return concrete_ptr->getValue();
    }
    else {
        return defaultValue;
    }
}
```

In this way, using a combination of polymorphism and templates, parameters of any type can be inserted into and extracted from a ParameterList.

5.3 Parallelization

The MUMPS library enables parallelization through MPI and we decided to keep this feature in our interface MumpsSparse. Our aim was to make the use of our class as easy as possible for the user: in fact to enable parallelization it is only necessary to initialize MPI with the function MPI_Init and to set the parameter nproc to the desired number of parallel processes, using the method
setParameters described in the previous section. This is accomplished with the following steps:

1. When an instance of MumpsSparse is set up, it spawns the processes needed for the parallel computation through the MPI function
   
   ```
   int MPI_Comm_spawn(const char *command,
                      char *argv[],
                      int maxprocs,
                      MPI_Info info,
                      int root,
                      MPI_Comm comm,
                      MPI_Comm *intercomm,
                      int array_of_errcodes[])
   ```

   which spawns (at most) maxprocs instances of program command in the runtime system defined in info.

2. The output parameter intercomm contains the intercommunicator between the original group and the newly spawned group. Through the MPI function MPI_Intercomm_merge, this intercommunicator is merged into a C intracommunicator, which is in turn translated into a Fortran communicator, used to set up the parameter comm_fortran in the MUMPS instances, both in the master and in the slave processes. These steps are necessary to ensure a correct functioning of the MUMPS library functions, as it only accepts intracommunicators.

3. The spawned processes can be thought as slave processes, which wait for instructions from the spawning master process: when MumpsSparse::factorize or MumpsSparse::solve are called, the master process broadcasts the appropriate job id to the slave processes and the computation starts. When the computation ends, the slave processes go back to wait for the next instruction from the master.

When the R interface is used, to enable the parallelization, the user needs to initialize MPI through the R package Rmpi and may also set the number of processors and specify the hostfile to be used for the computation.

5.4 Example of application

For the sake of clarity we provide an example of usage of SpLinearSolvers, for example using MumpsSparse, in order to solve $Ax = b$. 
int n = 100;
Eigen::SparseMatrix<double> A(n, n);
Eigen::VectorXd b(n);
// Filling the matrix and right-hand side
...
// Definition of a list of parameters for the solver
LinearSolvers::ParameterList list;
list.set("icntl[14]", 100);
list.set("sym", 2);
list.set("par", 1);
list.set("nproc", 2);
// Instantiate a MumpsSparse solver
LinearSolvers::MumpsSparse solver(list);
// Invoke the factorization on matrix A
solver.factorize(A);
// Solve the system Ax = b system and retrieve x
solver.solve(b);
auto x = solver.getSolution();
Chapter 6

Tests

The source code of the following tests can be found in the directory tests and, to launch them, R should be invoked from this directory.

6.1 Test coherence of the stochastic estimator

The goal of this first test is to check the behaviour of the stochastic estimator’s intrinsic error. As it was discussed in the first chapter, the computation of the Generalized Cross Validation is made in order to chose the best value of the parameter λ, that is the one minimizing the GCV. Therefore we are interested in testing whether the calculation with the stochastic approximation would affect this choice.

For this purpose we run several tests over a wide range of values for λ, (λ ∈ [10^{-3}, 10]), having a grid of 900 elements and possibly 4 covariates. Taking in consideration both the case of no covariates and location on nodes as well as location not on nodes with covariates, we plot the results of the exact computation against the stochastic estimator. The stochastic estimator is computed with 1000 realization, which was found to be a good compromise between speed and accuracy of the computation.

From the plots in 6.1 we can see that the stochastic estimator is greatly satisfying from this point of view: the plots in both cases are shaped equally, so the chosen parameters would be the same.
Figure 6.1: Case: no covariates, location on nodes

Figure 6.2: Case: covariates, location not on nodes
6.2 Test on variance

This second test is focused on the variance of the stochastic estimator, which we want to be sure is decreasing when the number of realizations increases. We want to see in different scenarios (with a different number of realizations) what is the magnitude of the variance and if it keeps the expected behavior. We are able to calculate n independently generated estimators of the edf and since we know that, when $\hat{edf}_i$ are independent, it holds:

$$Var[edf^{(n)}] = Var \left[ \frac{\hat{edf}_1 + \hat{edf}_2 + \cdots + \hat{edf}_n}{n} \right] = \frac{1}{n} Var[\hat{edf}_i]$$

we want to verify it. To estimate $Var[edf^{(n)}]$ we perform m realizations of $edf^{(n)}$ and we compute the sample variance. To allow the m realizations to be independent, we must keep track of the state of the random number generator. This is achieved through the parameter RNGstate.

In figure 6.2 are the results with $m = 200$ plotted against the expected result on a loglog graph; we can see that the variance has the proper behaviour.

![Figure 6.3: Test 2: variance decreasing with the number of realizations](image)

Figure 6.3: Test 2: variance decreasing with the number of realizations
6.3 Test on scalability

The aim of this test is to see if the parallel version of MUMPS improves the performances in the computation of the edf. We performed the test on Gigat, a cluster which allows the simultaneous usage of 2 computational nodes with 32 processors each. The settings are similar to the previous tests, covariates are present and the nodes are not located in the observations, and the GCV is computed stochastically. In this case the number of nodes is higher and equal to 90000 and of course the solver chosen is MumpsSparse. In Table (6.1) we can find the time required by the computation when the number of processors varies. As we can see the scalability is consistent at the beginning but it doesn’t improve much when the number of processors is higher than 8.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time [s]</td>
<td>81</td>
<td>54</td>
<td>41</td>
<td>38</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 6.1: Test on scalability
6.4 Performance comparison: Exact VS Stochastic

This last test aims at comparing the computational requirements of the exact and the stochastic approaches. The test is done for the case of location not on nodes, with four covariates and an increasing number of nodes in the grid, going from 400 to 4900. For the stochastic estimator a number of 500 realizations has been chosen. In table (6.4) it can be seen that with smaller grids the computational times are almost equivalent but as the size of the problem grows, the stochastic approach becomes more and more convenient and at the same time it retains an acceptable accuracy.

<table>
<thead>
<tr>
<th>$K$</th>
<th>Computed edf</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exact</td>
<td>Stochastic</td>
</tr>
<tr>
<td>400</td>
<td>5.3182</td>
<td>5.3238</td>
</tr>
<tr>
<td>900</td>
<td>5.4943</td>
<td>5.4765</td>
</tr>
<tr>
<td>1600</td>
<td>5.6418</td>
<td>5.7193</td>
</tr>
<tr>
<td>2500</td>
<td>5.7670</td>
<td>5.7733</td>
</tr>
<tr>
<td>3600</td>
<td>5.9010</td>
<td>5.8417</td>
</tr>
<tr>
<td>4900</td>
<td>6.0117</td>
<td>5.8776</td>
</tr>
</tbody>
</table>

Table 6.2: Performance comparison between exact and stochastic approach
Bibliography


