Linear Models with Partial Differential Equations
Regularization for data distributed over 2D manifolds

Anna Cosmo, Mario Beraha
APSC Project
Contents

1 Introduction

2 Mathematical Model
   2.1 Mathematical Prerequisites
   2.2 Finite Element on Polygonal Surfaces
      2.2.1 Representation of the surface
      2.2.2 Finite Element Spaces
   2.3 Discretization of the Laplace-Beltrami Operator
   2.4 Implementation

3 Code Structure
   3.1 The handling of the Mesh Structure
      3.1.1 Mesh objects: points and edges
      3.1.2 Mesh objects: triangles
      3.1.3 MeshHandler
   3.2 Solving the approximation problem
      3.2.1 Finite Elements and Matrix Assembly
      3.2.2 Assembling and solving the system
   3.3 The R/C++ Interface: .Call

4 Results
   4.1 Simulation studies
   4.2 OpenMP Parallelization

5 Tutorial
   5.1 Installing rgl
   5.2 Installing devtools
Abstract

The usage of PDEs in statistics has a well established history of success. In particular, it is possible to use PDEs to incorporate problem-specific prior information about the spatial structure of the phenomenon under study formalized in terms of a governing PDE. This also allows for a very flexible modeling of space variation, that accounts naturally for anisotropy and non-stationarity. We propose a regression model for data spatially distributed over nonplanar two-dimensional Riemannian manifolds. The model is a generalized additive model with a roughness penalty term involving a suitable differential operator computed over the non-planar domain. Thanks to a semi-parametric framework, the model allows for inclusion of space-varying covariate information.
1 Introduction

We present an approach for modelling data distributed over non planar two-dimensional surfaces (or manifolds). Quite informally, we can define a 2-dimensional manifold as a "2D object" embedded in a 3D space. In particular this entails that, locally, the 2D manifold is homeomorphic to $\mathbb{R}^2$. For more detailed information, one can refer to [1].

![Illustrative example of a two-dimensional manifold](image)

**Figure 1** – An illustrative example of a two-dimensional manifold

Many models are available to deal with spatial data over planar domains, but only few can deal with data distributed over manifolds. The most recent ones ([3]), involve the construction of a flattening map that maps the manifold into a subspace of $\mathbb{R}^2$, solve the PDE over the planar domain and then map the results back to the original manifold. Since the spatial regression model over planar domains is combined with the flattening map, it is not suited to deal with domains presenting complicated shapes such as carotid arteries with large aneurisms as multiple points on the wall of the superficial mesh would be mapped to the same point in the plane ([3]). In our approach, instead, we solve the equation directly on the manifold.
2 Mathematical Model

Consider \( n \) fixed data locations \( \{ x_i = (x_{i1}, x_{i2}, x_{i3}) : i = 1, \ldots, n \} \) lying on a non-planar domain \( \Sigma \) that is a uniformly regular surface embedded in \( \mathbb{R}^3 \). For each location \( x_i, \) a real-valued random variable of interest, \( z_i, \) is observed. We assume the model

\[
\begin{align*}
  z_i &= f(x_i) + \varepsilon_i \quad i = 1, \ldots, n
\end{align*}
\]

where \( \varepsilon_i, \ i = 1, \ldots, n, \) are independent observational errors with zero mean and constant variance \( \sigma^2, \) and \( f \) is a twice continuously differentiable real-valued function defined on the surface domain \( \Sigma. \) Our goal is to estimate the function \( f \) in \( 1. \) We propose to estimate \( f \) with \( \hat{f}, \) that is the function minimizing the following penalized sum of squared error functional

\[
J_{\Sigma, \lambda}(f) = \sum_{i=1}^{n} (z_i - f(x_i))^2 + \lambda \int_{\Sigma} (\Delta_{\Sigma} f(x))^2 \, d\Sigma
\]

where \( \Delta_{\Sigma} \) is the Laplace-Beltrami operator for functions defined over the surface \( \Sigma. \)

As shown in [4], this fourth order problem can be split into a coupled system of two second order equations as follows:

\[
\begin{align*}
\sum_{i=1}^{n} (\hat{z}_i - \hat{f}(x_i)) + \lambda \int_{\Sigma} (\nabla_{\Sigma} u \nabla_{\Sigma} g) \, d\Sigma &= 0 \\
\int_{\Sigma} u \, d\Sigma + \int_{\Sigma} (\nabla_{\Sigma} v \nabla_{\Sigma} \hat{f}) \, d\Sigma &= 0
\end{align*}
\]

for all \( (u, v) \in (H^1_{\text{int}}(\Sigma) \cap C^0(\Sigma)) \times H^1(\Sigma) \)

It is also possible to include in the model some other information measured at the same locations \( \{ x_i, i = 1, \ldots, n \}, \) as covariates of the linear model. In this case the functional to minimize becomes:

\[
J_{\Sigma, \lambda}(f) = \sum_{i=1}^{n} (z_i - w_i^T \beta - f(x_i))^2 + \lambda \int_{\Sigma} (\Delta_{\Sigma} f(x))^2 \, d\Sigma
\]

In order to be able to define the Laplace-Beltrami and the gradient operator on our manifold, provide a weak formulation of the problem and define an integral over this surface, we need a little bit more of theory, we mainly refer to [7].

2.1 Mathematical Prerequisites

Definition Parametric Surfaces. For \( d \geq 2 \) and \( 1 \leq n \leq d \) we call \( \Sigma \subset \mathbb{R}^{d+1} \) a \( n \)-dimensional parametrized surface if for every \( x_0 \in \Sigma \) there exists an open set \( \mathcal{U} \subset \mathbb{R}^{d+1} \) with \( x_0 \in \mathcal{U}, \) an open connected set \( \mathcal{V} \subset \mathbb{R}^n \) and a bijective map \( \chi : \mathcal{V} \to \mathcal{U} \cap \Sigma \) with rank \( \nabla \chi = n \) on \( \mathcal{V}. \) The map \( \chi \) is called a local parametrization of \( \Sigma \) and the couple \( (\mathcal{U}, \chi) \) is called a chart. An atlas is a collection of charts \( \{ (\mathcal{U}_i, \chi_i) \}_{i \in I} \) such that \( \bigcup_{i \in I} (\mathcal{U}_i \cap \Sigma) = \Sigma. \)
From now on \( d = n = 2 \), so we will consider only surfaces embedded in \( \mathbb{R}^3 \).

**Definition**. The first fundamental tensor is a symmetric and positive definite matrix \( G \in \mathbb{R}^{2 \times 2} \), \( G = \{ g^{ij} \} \) defined by
\[
G := \nabla \chi^t \nabla \chi
\] (5)

Let \( q = \sqrt{\det G} \), we can now define the notion of integral over (a subset of) \( \Sigma \). Given a subdomain \( \mathcal{U}' \subset \mathcal{U} \) and a function \( v : \mathcal{U}' \cap \Sigma \to \mathbb{R} \), let \( \mathcal{V}' = \chi^{-1}(\mathcal{U}' \cap \Sigma) \) and \( v^* : \mathcal{V}' \to \mathbb{R} \) be the pullback of \( v \) defined as
\[
v^*(y) = v(\chi(y)) \quad \forall y \in \mathcal{V}'
\] (6)

Thus we define
\[
\int_{\mathcal{U}' \cap \Sigma} v = \int_{\mathcal{V}'} v^* q
\] (7)

In order to obtain the integral over the entire \( \Sigma \) it is sufficient to sum over the chart of an atlas of \( \Sigma \) multiplied times a normalizing constant. (Please refer to [7] Chapter 2.2 for the formula)

We first define the tangential gradient and divergence operators, and next combine them to introduce the surface - Laplace-Beltrami - operator.

**Definition**. The tangential (or surface) gradient is defined for a function \( v : \Sigma \to \mathbb{R} \) at \( x \in \Sigma \) as
\[
\nabla_{\Sigma} v(x) = \sum_{i,j=1}^{2} g^{ij} \partial_j v^*(y) \partial_i \chi^t(y) = \nabla v^*(y) G^{-1} \nabla \chi^t
\] (8)

In view of the expression of the tangential gradient, it is easy to define the tangential divergence as
\[
\text{div}_{\Sigma} v(x) = \text{Trace}(\nabla_{\Sigma} v(x)) = \sum_{i,j=1}^{2} g^{ij} \sum_{k=1}^{3} \partial_j v^*_k(y) \partial_i \chi_k(y)
\] (9)

Eventually, we define the Laplace-Beltrami operator (or surface Laplacian):
\[
\Delta_{\Sigma} v = \text{div}_{\Sigma}(\nabla_{\Sigma} v)
\] (10)

It is possible to prove (see [7]) that the divergence theorem holds also for functions defined over surfaces, so it can be possible to integrate by parts and obtain the weak formulation of the Laplacian. Indeed, let \( f, g : \Sigma \to \mathbb{R} \) be of class \( C^2 \), the following identity holds
\[
-\int_{\Sigma} \Delta_{\Sigma} f g = \int_{\Sigma} \nabla_{\Sigma} f \nabla_{\Sigma} g - \int_{\partial \Sigma} g \nabla_{\Sigma} f (\tau \times \nu)
\] (11)

Where \( \tau \) and \( \nu \) are, respectively, the tangent and normal vector.
2.2 Finite Element on Polygonal Surfaces

2.2.1 Representation of the surface

We assume that there exists a Lipschitz homeomorphism \( P_0 : \Gamma_0 \rightarrow \Sigma \subset \mathbb{R}^3 \) describing our surface \( \Sigma \) as a deformation of a 2 dimensional polyhedral surface \( \Gamma_0 \). The vinculum notation is to emphasize that \( \Gamma_0 \) is piecewise affine, indeed it is composed of \( M \) (closed) faces \( \Gamma_i, i = 1, ..., M \). Therefore we denote by \( P_i : \Gamma_i \rightarrow \mathbb{R}^3 \) the restriction of \( P_0 \) to \( \Gamma_i \).

We can finally induces a partition of \( \Sigma \) setting

\[
\Sigma^i := P_i^0(\Gamma_i^0)
\]

We assume that there is a reference simplex \( \hat{T} \subset \mathbb{R}^2 \) and an affine map \( \hat{X}^i_0 : \mathbb{R}^2 \rightarrow \mathbb{R}^3 \) such that \( \Gamma_i^0 = \hat{X}^i_0(\hat{T}) \) (see figure below for a more intuitive representation of these maps). Thus we can define \( \chi^i := P_i^0 \circ \hat{X}^i_0 : \hat{T} \rightarrow \Sigma^i \). The collection of these parametrizations is denoted as \( \chi = \{\chi_i\}_{i=1,...,M} \).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{map.png}
\caption{Example of the map for \( d=2 \)}
\end{figure}

Let us consider now a function \( v^i : \Sigma^i \rightarrow \mathbb{R} \), then the functions \( \hat{v}^i : \hat{T} \rightarrow \mathbb{R} \) and \( \bar{v}^i : \Gamma_i^0 \rightarrow \mathbb{R} \) are defined uniquely via the maps \( \chi^i \) and \( P_0 \), namely

\[
\hat{v}^i(\hat{x}) := v^i(\chi(\hat{x})) \quad \forall \hat{x} \in \hat{T} \quad \text{and} \quad \bar{v}^i := v^i(P_i^0(\bar{x})) \quad \forall \bar{x} \in \Gamma_i^0
\]

Conversely, a function \( \hat{v}^i : \hat{T} \rightarrow \mathbb{R} \) (respectively, \( \bar{v}^i : \Gamma_i^0 \rightarrow \mathbb{R} \) defines uniquely the two functions \( v^i \) and \( \hat{v}^i \) (respectively, \( v^i \) and \( \hat{v}^i \)).

We can finally observe that, as in many real applications, we do not have access to a parametrization of the real \( \Sigma \), thus from now in our problem on \( \Sigma^i \) will coincide with \( \Gamma_i^0 \). We will consider \( \Sigma \) as a triangulated surface, that is the approximation of the real surface produced by some imaging technique. Hence we use directly \( \Gamma_0 \) and we actually do not know neither the map \( P_0 \) nor the exact surface.
2.2.2 Finite Element Spaces

We define the global quantity:
\[ \tau := \bigcup_{i=0}^{M} \Gamma_i^0 = \Gamma_0 \] (14)

We can now introduce the space \( \hat{V}_r := \hat{V}_r(T) \) which is the finite element space of polynomials of degree \( r \) \( (r = 1, 2) \) on \( T \). We use this space to easier compute matrices needed to solve our problem. We finally define
\[ \mathbb{V}_r(\tau) := \{ V \in C^0(\Gamma_0) \mid V |_{\Gamma_0} \text{ is the lift of some } \hat{V}^i \in \hat{V}_r \text{ via } X_0(T) \} \] (15)

2.3 Discretization of the Laplace-Beltrami Operator

Given \( f : \Sigma \rightarrow \mathbb{R} \) we want to derive a finite element approximation of the solution \( u : \Sigma \rightarrow \mathbb{R} \) of
\[ \begin{align*}
-\Delta_\Sigma u &= f \quad \text{on } \Sigma \\
u &= 0 \quad \text{on } \partial \Sigma
\end{align*} \] (16)

Thanks to (11) we can derive the weak formulation as
\[ u \in H^1_0(\Sigma) : \int_{\Sigma} \nabla_\Sigma u \nabla_\Sigma v = \int_{\Sigma} f v \quad \forall v \in H^1_0(\Sigma) \] (17)

Where \( H^1_0(\Sigma) = \{ v \in H^1(\Sigma) : v = 0 \text{ on } H^1_0(\Sigma) \} \), and \( f \in L^2(\Sigma) \).

The approximation of the solution to a PDE defined over a surface depends on

- how well the surface can be approximated;
- how well the solution can be approximated when mapped to an approximate surface.

The discrete variational problem treated in our work is defined over an approximate surface, therefore leading to a discrete problem which is only weakly consistent.

2.4 Implementation

Considering a physical triangle \( K \in \tau \), the following equivalence holds:
\[ \nabla_K v(x) = \nabla \hat{v}(y) G_\Gamma^{-1}(\nabla \hat{X}_0)^t \text{ for } x \in K \quad \text{and} \quad y \in \hat{T} = \hat{X}_0^{-1}(K). \] (18)

Here \( G_\Gamma := (\nabla \hat{X}_0)^t \nabla \hat{X}_0 \) is the metric corresponding to \( \Gamma \). We deduce
\[ \int_K \nabla_K U \nabla_K V = \int_{\hat{T}} \nabla \hat{U} G_\Gamma^{-1}(\nabla \hat{X}_0)^t \nabla \hat{X}_0 G_\Gamma^{-1} \nabla \hat{V} Q_\Gamma = \int_{\hat{T}} \nabla \hat{U} G_\Gamma^{-1} \nabla \hat{V} Q_\Gamma \] (19)

where \( Q_\Gamma := \sqrt{\det G_\Gamma} \) is the area element corresponding to \( \Gamma \).

Note that \( G_\Gamma \) and \( Q_\Gamma \) are piecewise constant.
3 Code Structure

In the implementation of an R library able to treat 2D manifolds embedded in a 3-dimensional space, we can identify three different core issues that we need to solve:

- the handling of the mesh structure;
- the resolution of the approximated problem;
- the R/C++ interface.

In this section we will explain how we implemented these three different sides of the problem and the reasons of our choices. We will use an inductive approach, indeed, to introduce some more complex structures, we first have to describe the simpler ones. Before entering in the details of the code, we specify that the leading idea of our work is to maintain the possibility of using the 2D domain without changing anything in the data and the mesh passed to the fdaPDE library.

Due to an initial choice made by the first developer of the C++ code, we will use a slightly different syntax, indeed from now on we define:

```cpp
typedef double Real;
typedef int UInt;
```

3.1 The handling of the Mesh Structure

3.1.1 Mesh objects: points and edges

The mesh objects are all derived as son of the father class `Identifier`, which includes only two attributes (`Id, bcId`) and the methods to return them. From this class we inherit the classes `Point, Edge and Triangle`.

```cpp
class Point: public Identifier{
    private:
        std::vector<Real> coord;
    public:
        UInt ndim;
        ...
}
```

This class implements a basic C++ object that represents a point. `ndim` defines if we are considering points in a 2D or 3D domain.

The Point class is then used to construct the following class

```cpp
class Edge: public Identifier{...}
```

that did not require any modification from the previous version of the library.
3.1.2 Mesh objects: triangles

The first important decision we had to make, was how to modify the class Triangle in order to be able to deal both with triangles laying on a 2D plane and on a surface embedded in a 3D space. This choice highly impacts on the whole structure of the code, as it will be clear with class FiniteElement. Due to this very reason, we decided to use template specialization in order to be able to keep the structure of the code developed for the planar domain almost identical to the original version and to provide flexibility for future developers that may want to extend the code (for example to volumetric finite elements).

We added to the classes two template parameters: ndim that specifies the dimension of the space the objects are embedded into, and mydim, that quantifies the "local dimension" of the object considered (for example, in our case ndim=3, mydim=2).

![Inheritance graph for class Triangle (Doxygen)](image)

Figure 3 – Inheritance graph for class Triangle (Doxygen)

We notice that this choice is particularly suited for our problems, indeed the approach needed to solve the problem on a manifold is strongly different from the one used to solve the planar problem. Inside the class of a 2D triangle we didn’t change anything from the original fdaPDE, so we will expose the structure of 3D element.

```cpp
template <UInt NNODES>
class Triangle<NNODES,2,3> : public Identifier {
    private:
        std::vector<Point> points_;
        Eigen::Matrix<Real,3,2> M_J_;
        Eigen::Matrix<Real,2,2> G_J_;
        Eigen::Matrix<Real,2,2> metric_;
        Real detJ_;  
        void init(const std::vector<Point> &points);
        ...  
}
```

Comparing this class with the 2D case it is possible to notice the lack of the matrix
$M_{inv \cdot J}$ and the deep difference in the construction of the other matrices (see the method `init`).

Let us first clear up the theoretical meaning of those attributes. As we exposed in the subsection 2.2.2, we need a map going from the reference 2D triangle $\hat{T}$ to the 3D triangle we are working on (see figure 4).

![Figure 4 – Reference triangle](image)

Letting $K_i$ be an element of the approximated manifold, the map $X_i^0 : \hat{T} \rightarrow K_i$ is:

$$x = M^i J \hat{x} + b^i \quad \forall \ x \in K_i$$

where

$$M^i J = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \\ z_2 - z_1 & z_3 - z_1 \end{bmatrix}$$

and

$$b^i = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$$

(20)

The method `init` computes this matrix that is stored in the attribute $M \cdot J$ (notice that this coincides with the gradient of the map). Accordingly to this definition of the map $X_i^0$, in attribute $G \cdot J$ is stored what we defined earlier in section 2.4 as $G_T$, while $\text{metric}_\cdot$ contains $G_T^{-1}$.

Another important method inside this class is `isPointInside`, that given a `Point`, checks if this point is inside the triangle. To do so, we solve 3 scalar equation in 2 unknowns: $(u, v)$. Let $P_0, P_1, P_2$ be the 3 vertices of the triangle (each of them has 3 components $x, y, z$), $P$ the point that needs to be checked. The system is

$$(P_1 - P_0)u + (P_2 - P_0)v = P - P_0$$

(21)

. If the system is solvable, than $P$ is in the same plane as triangle $(P_1, P_2, P_0)$, if in addition

$$u, v \geq 0 \quad \text{and} \quad u + v \leq 1$$

(22)

then $P$ is inside the triangle.

Another important function, `evaluate_point`, is needed. Its aim is to evaluate the solution in different points from the ones considered in the resolution. With the templates for the dimension, we extend this function also for the case of a manifold domain which requires some different tricks.
template <UInt ORDER, UInt mydim, UInt ndim>
inline Real evaluate_point(const Triangle<3*ORDER, mydim, ndim>& t, const Point& point, const Eigen::Matrix<Real, 3*ORDER, 1>& coefficients) {...};

A similar function, evaluate_der_point, was previously implemented in order to evaluate the derivatives, but our work does not require a templatization of this one thanks to the usage of the reference triangle.

3.1.3 MeshHandler

The class MeshHandler previously implemented for the 2D domain was basically an interface with the C library Triangle, that triangulates the domain. Differently, we assume that the mesh is passed by R through two matrices, one contains the vertices and the other one the connectivity. As a consequence, we changed MeshHandler taking advantage of the template structure. To do so, a special constructor is needed; it takes as an input a SEXP (for the details, refer to section 3.3) that represents the mesh as stored in R, and stores the matrices in two std::vector.

```cpp
#ifdef R_VERSION_
template <UInt ORDER>
MeshHandler<ORDER, 2, 3>::MeshHandler(SEXP mesh)
{
    mesh_ = mesh;
    num_nodes_ = INTEGER(VECTOR_ELT(mesh_, 0))[0];
    num_triangles_ = INTEGER(VECTOR_ELT(mesh_, 1))[0];
    points_.assign(REAL(VECTOR_ELT(mesh_, 2)),
                   REAL(VECTOR_ELT(mesh_, 2)) + 3*num_nodes_);
    triangles_.assign(INTEGER(VECTOR_ELT(mesh_, 3)),
                      INTEGER(VECTOR_ELT(mesh_, 3)) + 3*ORDER*num_triangles_);
    std::for_each(triangles_.begin(), triangles_.end(),
                  [](int& i){i-=1;});
}
#endif
```

Observe that is needed a conversion of the indices, indeed indexing starts from 0 in C++, whereas it starts from 1 in R.

The class also contains several methods (getPoint(Id id), getTriangle(Id id), ...) that provide an object-oriented interface to the data structure shown.
3.2 Solving the approximation problem

The solution of problem 3 obviously requires some changes in the code, in particular the parts of the fdaPDE library regarding the resolution of the PDEs.

3.2.1 Finite Elements and Matrix Assembly

Every finite element contains 2 triangles: the 2D reference triangle, and one of the 3D ones in the mesh. This is because almost any operation needed to solve our problem is done mapping the generic 3D "physical" triangle we are working on to the reference one.
The physical triangle is then uploaded by the method
\texttt{FiniteElement<Integrator, ORDER,2,3>::updateElement} during the iterations over
the elements of the mesh. The other methods of this class allow us to return all the quan-
tities required for the iteration, in particular the evaluations of the basis on quadrature
nodes.

Now we can describe how matrices are built. The technique is the one described in [5],
where expression templates are used to implement Galerkin methods.

\texttt{class Assembler}\{\par
\texttt{public:\par}
\texttt{template<UInt ORDER, typename Integrator, typename A>\par}
\texttt{static void operKernel(EOExpr<A> oper,\par}
\texttt{const MeshHandler<ORDER,2,3>& mesh,\par}
\texttt{FiniteElement<Integrator, ORDER,2,3>& fe, SpMat& OpMat);}\par
\texttt{template<UInt ORDER, typename Integrator>\par}
\texttt{static void forcingTerm(const MeshHandler<ORDER,2,3>& mesh,\par}
\texttt{FiniteElement<Integrator, ORDER,2,3>& fe,\par}
\texttt{const ForcingTerm& u, VectorXr& forcingTerm);}\par
\}

This class, \texttt{Assembler}, discretizes the generic differential operator defined by \texttt{class A.}
In particular the method \texttt{operKernel} translates the finite element resolution into an
algebraic system, defined through the template operation \texttt{A}. This structure allows a simple
definition of the different types of matrices we use, and eventually others if required in
the future. For our purposes, \texttt{class A} can be either

\texttt{class Mass}\{\par
\texttt{public:\par}
\texttt{...\par}
\texttt{template<class Integrator ,UInt ORDER>\par}
\texttt{inline Real operator() (FiniteElement<Integrator,\par}
\texttt{ORDER,2,3>& currentfe_, UInt i, UInt j,\par}
\texttt{UInt iq, UInt ic = 0)}\par
\texttt{...\par}
\}

or

\texttt{class Stiff}\{\par
\texttt{public:\par}
\texttt{...\par}
\texttt{template<class Integrator, UInt ORDER>\par}
\texttt{inline Real operator() (FiniteElement<Integrator,\par}
\texttt{ORDER,2,3>& currentfe_, UInt i, UInt j,\par}
\texttt{UInt iq, UInt ic = 0)}\par
\texttt{...\par}
\}
Where \texttt{operator(...) \ldots} is responsible for the integration of the basis functions on the element \texttt{currentfe}.

### 3.2.2 Assembling and solving the system

This final part is substantially implemented in the class

\begin{verbatim}
template<
  typename InputHandler, typename Integrator, UInt ORDER,
  UInt mydim, UInt ndim>
class MixedFERegression{};
\end{verbatim}

Here the code did not actually require any type of variation, but for the addition of templates \texttt{mydim,ndim}.

The resulting Call Graph is thus the following

![Call Graph](image)

\textbf{Figure 7} – Call Graph for class MixedFERegression (Doxygen)

To finally solve our approximation problem, we are only left to explain how this library is linked with R and how data is passed.

### 3.3 The R/C++ Interface: .\texttt{Call}

The interface used to let R communicate with the C++ source code, is the one described in [6], namely .\texttt{Call}. It is a powerful, yet quite simple, way to use precompiled code in R.

To explain the syntax, let us briefly look into this simple example: we want to call a C++ function from R in order to sum up two numbers.
In C++

```c
#include <R.h>
#include <Rinternals.h>

extern "C"{
    SEXP add(SEXP a, SEXP b) {
        SEXP result = PROTECT(allocVector(REALSXP, 1));
        REAL(result)[0] = asReal(a) + asReal(b);
        UNPROTECT(1);
        return result;
    }
}
```

This file needs to be compiled with `R CMD SHLIB` in order to create a shared object that can be dynamically loaded in R:

```r
dyn.load("mysharedlib.so")

add <- function(a, b) {
    .Call("add", a, b)
}
```

In particular, notice that all R objects are seen by C++ as SEPs, that stands for S-Expression. This may lead to some confusion especially when R objects have many attributes, thus some care is needed.

Moreover, observe that `.Call` provides an interface between C and R, so that also in our code, an `extern "C"` is needed.

For bigger projects (with multiple source C++ files) it is convenient to include in the `src` directory a file named `Makevars`, where all the flags needed for the compiling and linking are stored. This file needs to be portable since the ultimate goal is to distribute the project (through a package) to other users. To do so many options are available, we refer to [6] (Section 1.6 and 1.2.1).

Once the source code and the R user interface are ready, the following step is to create and install a R package. We will briefly discuss the options available in section 5.

For our purposes, the main tasks needed to be done by R are:

- Passing the mesh to C++
- Passing the data on which to perform the regression to C++
- Calling the solver
- Graphical plots

The first two are easily solved by implementing special constructors in the classes
MeshHandler and RegressionData, taking care of the indexing difference between R and C++.

The third question the most delicate one and is the real heart of the whole interface of the R package. It is addressed by the C++ file "fdaPDE.cpp", where all the various options are handled and the various C++ classes are created accordingly, and from the various R files in the subdirectory "R".

To sum up everything, the workflow behind a call of the R function smooth.FEM.basis(...) will look like:

\begin{description}
\item[Call of the R function] 
\texttt{output = smooth.FEM.basis(...)}
\item[R code execution] 
- Check parameters 
- Set type for correct C++ reading 
- Call C++ code using \texttt{.Call} 
  \item[C++ code execution] 
  - Check on the template parameters ndim, mydim, order 
  - Build C++ objects using special constructors from R objects 
  - Solve the approximation problem 
  - Save results in R memory 
  - Create R objects to be returned 
\end{description}

For the graphical plots, we relied upon the powerful R package \texttt{rgl} to implement a function that plots both the triangulated surface and a colormap over it according to the nodal values of the solution. In order to give the possibility of using the second order approximation, we also include an R function which creates a second order mesh from a first order one.
4 Results

Firstly, we want to assess the performance of our implementation against the one described in [3], both in terms of accuracy and computational cost.

We notice a big difference in the approach to the approximation problem. Indeed, while our algorithm solves the PDE directly on the manifold, the latter involves the construction of a flattening map, \( G : \Sigma \to \Omega \), where \( \Sigma \) is the manifold and \( \Omega \) a suitable subset of \( \mathbb{R}^2 \).

In the following comparisons we decided to focus only on the model without covariates, since the main contribution of our work is related only on the penalization term. Thus, although the linear regression with covariates has shown a good accuracy, we will not present those results.

The tests were performed on a AMD FX(tm)-4300 Quad-Core Processor @ 3.8GHz with 8Gb RAM.

4.1 Simulation studies

In this section we illustrate the performance of the proposed technique on different non-planar domains. We compare two methods:

- the proposed spatial regression model over non-planar domains (fdaPDE)
- the spatial regression model presented in [3] (ConfFlat)

The two methods are compared on a test domain (the second on in figure 8).

The data was generated in the following way:

- extract \( a_i \) independently from a normal distribution with mean and variance equal to one
- \( y_i = f_i + \varepsilon_i \), where \( \varepsilon_i \sim N(0, \sigma^2) \) (i.i.d)

Where \( f_i = f(x_i, y_i, z_i) \) \( i = 1, \ldots \text{num nodes} \) where \( (x_i, y_i, z_i) \) are the coordinates of the \( i \)-th node and \( f \) can be of the form

- \( f(x_i, y_i, z_i) = a_1 \sin(2\pi x_i) + a_2 \sin(2\pi y_i) + a_3 \sin(2\pi z_i) + 1 \)
- \( f(x_i, y_i, z_i) = a_1 \log(x_i + 2) + a_2 \log(y_i + 2) + a_3 \log(z_i + 2) \)
- \( f(x_i, y_i, z_i) = a_1 \exp(a_2 x_i) + a_3 \exp(a_4 y_i) + a_5 \exp(a_6 z_i) \)
- \( f(x_i, y_i, z_i) = a_1 (x_i + 2)^{a_2} + a_3 (y_i + 2)^{a_4} + a_5 (z_i + 2)^{a_6} \)
And the variance of the normal error $\sigma$ is equal to 10% of $|\max(f_i) - \min(f_i)|$ (on average).

Finally, for these methods the optimal value of the smoothing parameter $\lambda$ is selected at each simulation replicate and for each domain by generalized cross validation.

**Figure 8** – Simulation study. Left: colormap of a test function. Middle: data with noise. Right: estimates provided by the spatial regression model (fdaPDE).
As we can see, the MSE boxplots (figure 9) show a greater accuracy using our approach for every proposed choice of the function $f$. Most importantly, we can see a really significant computational speedup (figure 10). This is mostly due to two factors: we are using compiled source code while in [3] the algorithm was implemented in Matlab, the construction of the flattening map is computationally intensive (actually more than solving the PDE system on the flattened domain).

It must be specified that in the comparison of the computational costs, the GCV was not computed both in our code and in ConfFlat. This is mainly because a newer approach for its computation is under development (it is a current PACS Project) and this will lead to a significant speedup in that task.

Another interesting comparison to be done is the one regarding the order of the finite element basis used. For this task, we generated data as above, located on the barycenter of each triangle of the mesh using the first choice of function $f$ and added a gaussian
Figure 11 – Comparison of MSE using FE basis of order 1 and 2 on the first and second geometries of figure 8.

noise with standard deviation 1.5. In figure 11 the results are shown. We can see that in both the geometries the enhancement of the performance obtained using quadratic finite elements is not really significative.

4.2 OpenMP Parallelization

Since setting the right penalization term $\lambda$ is one of the critical choices to make when developing a statistical model, it is possible for the user to pass to the C++ code a vector $\lambda$ with different choices of the parameter.

Given this vector, a different solution is computed for each of its components and then returned to the R user along with some statistics.

This is the real bottleneck of the code since this procedure, given a vector $\lambda$ with $q$ components, requires performing the resolution of a large linear system $(2 \times 2)$ exactly $q$ times.

However this is also highly parallelizable since the matrices involved in the linear system are only slightly different. So, with some light changes in the class MixedFERegression it was possible to introduce an OpenMP parallelization.

The median of the performances over fifteen trials for increasing number of cores with a vector of twelve values for $\lambda$ are plotted below.
**Figure 12** – Performance scaling on a mesh with 551 nodes and 1020 triangles
5 Tutorial

Once the source code is downloaded from https://github.com/mariob6/progetto_pacs/ and unzipped, there are two ways for installing the R package on your machine.

The first one requires the use of devtools, a popular R package, it will also take care of installing all dependencies and install the fdaPDE package in the same directories of all the other R packages installed. It suffices to run from the package root folder:

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

To build the documentation in Roxygen then:

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

This second command will create a subfolder named man where the .Rd files will be stored. This will be useful when calling for "help" from R (e.g. ?smooth.FEM.basis)

The second way to complete the installation does not require any additional package, but will throw errors if the packages required for the functioning of fdaPDE are not installed.

From the terminal, run:

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

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

Rscript <chosen-test.R>

or loading the script directly in R.

To build the Doxygen documentation of the C++ code instead, navigate to the src folder and type

doxxygen -g <config-file>

This will create a configuration file (if <config-file> is missing, it will be named Doxygen-file), that can be edited to customize the output.

To run doxygen then type:

doxxygen <config-file>

To get the Reference Manual, then go to the latex subfolder that has been generated and type make.
5.1 Installing rgl

In Ubuntu or other Linux distribution, it is quite common to have troubles in installing the R package rgl, this will also cause the installation procedure described above to fail and throw errors like:

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

or

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

The fix we propose is to install the following packages before installing rgl or fdaPDE:
xorg-dev libx11-dev mesa-common-dev libglu1-mesa-dev

5.2 Installing devtools

If the installation of the package devtools throws the following errors:

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

please try to install the following libraries:

- Ubuntu: `sudo apt-get install libcurl4-openssl-dev libssl-dev`
- CentOS: `sudo yum -y install libcurl libcurl-devel`
References


