APSC PROJECT
Academic year: 2017-2018

Kriging prediction for manifold-valued data
The Manifoldgstat R-package

Ilaria Sartori
Luca Torriani

Supervised by
Prof. Alessandra Menafoglio
Prof. Piercesare Secchi

21 December 2018
Contents

Abstract 2

1 The statistical problem 3
   1.1 Introduction ............................................. 3
   1.2 Positive definite symmetric matrices .................. 3
       1.2.1 Model and Kriging prediction ............. 4
   1.3 Correlation matrices ................................. 6
       1.3.1 Model and Kriging prediction ............. 8

2 Code structure 8
   2.1 Overview .................................................. 8
   2.2 Factories to handle run-time parameters ............ 12
   2.3 Gauss-Newton to compute variogram parameters ....... 15
   2.4 Some technical remarks .................................. 17
   2.5 Code documentation ..................................... 18

3 Interface R/C++ 18
   3.1 Debug ..................................................... 19

4 Manifoldgstat package 21

5 Results 22
   5.1 Positive definite symmetric matrices .............. 22
   5.2 Correlation matrices .................................... 24

6 Conclusion 27
   6.1 Future work ............................................... 28

7 Installation 28

References 30
Abstract

The statistical analysis of data belonging to Riemannian manifolds is becoming increasingly important in many applications, such as shape analysis, diffusion tensor imaging and the analysis of covariance matrices. In many cases, data are spatially distributed but it is not trivial to take into account spatial dependence in the analysis because of the non linear geometry of the manifold. This project focuses on the implementation of a slight variation of the solution to the problem of spatial prediction for manifold-valued data proposed in [1]. In that article only correlation matrices were considered, while here the analysis is also extended to correlation matrices.
1 The statistical problem

1.1 Introduction

Positive definite symmetric matrices, as for instance covariance and correlation matrices, are an important example of data belonging to a Riemannian manifold. Under the hypothesis that the dispersion of the observations on the manifold is not too large, a tangent space can be used to approximate data in a vector space, where a linear model can be employed to describe the relationship between the response variable and the covariates. This allows to extend well established statistical methods for regression models to the context of manifold-valued response variable.

1.2 Positive definite symmetric matrices

Let \( PD(p) \) indicate the Riemannian manifold of positive definite symmetric matrices of dimension \( p \) and let \( Sym(p) \) be the space of symmetric matrices of dimension \( p \). The tangent space \( T_\Sigma PD(p) \) to \( PD(p) \) in the point \( \Sigma \in PD(p) \) can be identified with the space \( Sym(p) \), which, unlike \( PD(p) \), is linear and can be equipped with an inner product. A Riemannian metric in \( PD(p) \) is then induced by the inner product in \( Sym(p) \). Each definition of distance has a corresponding geodesic and thus exponential and logarithmic map, which, respectively, map from \( Sym(p) \) to \( PD(p) \) and vice versa.

The two most common choices for the inner product on \( Sym(p) \) are the following:

- **Frobenius**: \( \langle A, B \rangle = \text{tr} (A^T B) \)
- **Frobenius Scaled**: \( \langle A, B \rangle_\Sigma = \text{tr} \left( \Sigma^{-\frac{1}{2}} A^T \Sigma^{-1} B \Sigma^{-\frac{1}{2}} \right) \)

As far as the Riemannian metrics in \( PD(p) \) are concerned, we implemented:

- **Frobenius Scaled**: \( d_F (P_1, P_2) = \| \log \left( P_1^{-\frac{1}{2}} P_2 P_1^{-\frac{1}{2}} \right) \|_F \)
- **Log-Euclidean**: \( d_L (P_1, P_2) = \| \log (P_1) - \log (P_2) \|_F \)
- **Square-Root**: \( d_S (P_1, P_2) = \| P_1^{\frac{3}{2}} - P_2^{\frac{3}{2}} \|_F \)

and the corresponding maps:

- **Frobenius Scaled**:
  \[
  \exp_\Sigma (A) = \Sigma^{\frac{1}{4}} \exp \left( \Sigma^{-\frac{1}{2}} A \Sigma^{-\frac{1}{2}} \right) \Sigma^{\frac{1}{2}}
  \]
\[ \log_{\Sigma} (P) = \Sigma^{\frac{1}{2}} \log \left( \Sigma^{-\frac{1}{2}} P \Sigma^{-\frac{1}{2}} \right) \Sigma^{\frac{1}{2}} \]

- **Log-Euclidean**:
  \[ \exp_{\Sigma} (A) = (\log(\Sigma) + A)^T (\log(\Sigma) + A) \]
  \[ \log_{\Sigma} (P) = \log (P) - \log (\Sigma) \]

- **Square-Root**:
  \[ \exp_{\Sigma} (A) = \left( \Sigma^{\frac{1}{2}} + A \right)^T \left( \Sigma^{\frac{1}{2}} + A \right) \]
  \[ \log_{\Sigma} (P) = P^{\frac{1}{2}} - \Sigma^{\frac{1}{2}} \]

where \( P, P_1, P_2, \Sigma \in PD(p) \) and \( A, B \in Sym(p) \).

### 1.2.1 Model and Kriging prediction

The raw data of the analysis are sampled from a non-stationary manifold-valued random field. The main idea is to use a tangent space model to approximate the geometry of the manifold and refer to the tangent space to deal with spatial dependence. Thus, the first step consists in computing the intrinsic mean of the data (see [3]) in order to estimate the tangent point \( \Sigma \). Then, the tangent space regression is applied. The proposed model is the following: for \( s \in D \subset \mathbb{R}^d \),

\[ S(s; \beta, \Sigma) = \exp_{\Sigma} (A(x(s); \beta) + \Delta(s)) \]  

where \( \Sigma \in PD(p) \) is the tangent point and \( A(x(s); \beta) \in Sym(p) \) is the regression prediction, being \( x \in \mathbb{R}^r \) the vector of covariates, \( \beta = (\beta_{0, \ldots, \beta_{r-1}}) \in Sym(p)^{r+1} \) the regression coefficients and \( r \) the number of predictors. \( \Delta \) is a zero-mean, globally second-order stationary and isotropic random field taking values in \( Sym(p) \). This means that \( \mathbb{E}[\Delta(s)] \) is the null matrix for all \( s \) in \( D \) and, for \( s_i, s_j \in D \), the covariance between \( \Delta(s_i) \) and \( \Delta(s_j) \) depends only on the distance between \( s_i \) and \( s_j \), i.e., \( \text{Cov}(\Delta(s_i), \Delta(s_j)) = C(\|s_i - s_j\|) \). Equivalently, under the previous assumptions, for \( s_i, s_j \in D \), the spatial dependence of the field can be represented by the semivariogram defined as \( \gamma(\|s_i - s_j\|) = \frac{1}{2} \text{Var}(\Delta(s_i) - \Delta(s_j)) = \frac{1}{2} \mathbb{E}[\|\Delta(s_i) - \Delta(s_j)\|^2_F] \).

In practice, if \( \beta \) were known, the spatial dependence of the random field \( \Delta \) on the tangent space could be estimated by considering the residuals \( \Delta(s_i) = \log_{\Sigma} (S_i) - A(x(s_i); \beta) \) and computing the empirical semivariogram as proposed in [5]:

\[ \hat{\gamma}(\|h\|) = \frac{1}{2 |N(\|h\|)|} \sum_{(s_i, s_j) \in N(\|h\|)} \|\Delta(s_i) - \Delta(s_j)\|^2_F \]  

(1.2)
where \( N(h) = \{(s_i, s_j) \in D : h - \Delta h < \|s_i - s_j\| < h + \Delta h; i, j = 1, \ldots, n\} \), being \( \Delta h \) a positive (small) quantity, \( h > 0 \) and \( |N(h)| \) the number of couples \((s_i, s_j)\) belonging to \( N(h) \). A model semivariogram \( \hat{\gamma}_m(h) \) can then be fitted to the empirical semivariogram, for example via weighted least squares (see Section 2.6.2 of [5]). In our work three possible choices are available for the model semivariogram: gaussian, exponential and spherical. Once the model semivariogram has been defined, an estimate of the covariogram can be obtained as \( \hat{C}\left(\|s_i - s_j\|\right) = \lim_{h \to \infty} \hat{\gamma}_m(h) - \hat{\gamma}_m(\|s_i - s_j\|) \).

The model for the symmetric matrix \( A \) has to be specified and, in this work, we consider a linear model
\[
A(x; \beta) = \sum_{k=0}^{r} \beta_{.k} Z_k \tag{1.3}
\]
where \( \beta_{.k} \in \text{Sym}(p) \) and the \( Z_k \) are the components of the vector \( Z = (1 \ x^T)^T \in \mathbb{R}^{r+1} \).

Let \( s_1, \ldots, s_n \), be distinct locations in the domain \( D \), where \( x(s_i) \) is observed together with a realization \( S_i \) of the random field (1.1). Let \( \hat{\Sigma} \) be the estimated tangent point and \( \hat{\Gamma} \) the estimated covariance matrix of the random errors in the observed locations, i.e. \( \hat{\Gamma}_{ij} = \hat{C}(\|s_i - s_j\|) \). The parameter \( \beta \) is computed as the generalized least square estimator in the tangent space:
\[
\begin{align*}
\hat{\beta}_{lq} \left( \hat{\Sigma} \right) & = \arg\min_{\beta_{lq} \in \mathbb{R}^{r+1}} \left( Z \beta_{lq} - Y_{lq} \left( \hat{\Sigma} \right) \right)^T \hat{\Gamma}^{-1} \left( Z \beta_{lq} - Y_{lq} \left( \hat{\Sigma} \right) \right) \\
& = \left( Z^T \hat{\Gamma}^{-1} Z \right)^{-1} Z^T \hat{\Gamma}^{-1} Y_{lq} \left( \hat{\Sigma} \right) \tag{1.4}
\end{align*}
\]
where \( Y_{lq}(\hat{\Sigma}) \) is the vector \( \left( (\log \hat{\Sigma}(S_1))_{lq}, \ldots, (\log \hat{\Sigma}(S_n))_{lq} \right)^T \) of elements in position \((l, q)\) of the projected observations.

To sum up, \( \Sigma \) can be estimated as the intrinsic mean of the \( S_i \), and then used to obtain estimates for \( \beta \) and \( \Gamma \) through an iterative application of (1.2) and (1.4). At this point a kriging interpolation of the residuals provides an estimate for the field \( S \) in the unobserved location \( s_0 \). Therefore, by applying the kriging theory in Hilbert spaces (see [2]), the simple kriging predictor for \( \Delta(s_0) \) is derived as \( \sum_{i=1}^{n} \lambda_i^0 \hat{\Delta}(s_i) \), where the vector of weights \( \lambda_0 = (\lambda_1^0, \ldots, \lambda_n^0)^T \) solves
\[
\lambda_0 = \hat{\Gamma}^{-1} c
\]
with \( c = \left( \hat{C}(\|s_1 - s_0\|), \ldots, \hat{C}(\|s_n - s_0\|) \right)^T \). Given the vector of covariates
observed at location $s_0$, the prediction of $S_0$ – the unobserved value of the field $S$ at $s_0$ – can be eventually obtained through the plug-in estimator:

$$\hat{S}_0 = \hat{S}_0(x(s_0), (x(s_1), S_1), \ldots, (x(s_n), S_n)) = \exp_{\mathbf{\Sigma}} \left( \hat{\beta}_{..0}(\hat{\Sigma}) + \sum_{k=1}^{r} \hat{\beta}_{..k}(\hat{\Sigma}) x_k(s_0) + \sum_{i=1}^{n} \lambda_i^0 \Delta(s_i) \right).$$

(1.5)

### 1.3 Correlation matrices

Let us now consider the space of correlation matrices. These form a subset of the space of $PD(p)$, due to the additional constraint of having ones on the diagonal. However, it is not possible to use the geometry of $PD(p)$ to deal with them. Indeed, the geodesic in $PD(p)$ between two correlation matrices is not bounded to be in the space of correlation matrices and, moreover, the local tangent approximation is not guaranteed to return valid correlation matrices. Therefore, we need to use a different space to carry out statistical analysis for correlation matrices. In particular, we can exploit the fact that the space of full rank correlation matrices of dimension $p$ can be identified with a specific set of upper triangular matrices, the Cholesky manifold, which is defined as follows:

$$Chol(p) = \{ H \in \mathbb{R}^{p \times p} : H_{ij} = 0 \text{ if } j < i; \ H_{11} = 1, \| H^{(i)} \|_{F(0)} = 1 \},$$

(1.6)

where $H^{(i)}$ denotes the vector of the non-zero element of the $i$-th column of $H$. This means that (i) if $H \in Chol(p)$, the $i$-th column of $H$ can be identified with an element of the $i$-th dimensional hypersphere $S^i$ with radius 1, and (ii) $Chol(p)$ is a product manifold with $Chol(p) = e_1 \otimes \prod_{i=2}^{p} S^i$, $e_1$ being the $p$-dimensional vector such that $e_{11} = 1$ and $e_{1i} = 0$ for $i = 2 \ldots p$. The key motivation to introduce this space is that there is a one-to-one correspondence between the Cholesky manifold and the space of correlation matrices, since for any $H \in Chol(p)$, $R = H^T H$ is a correlation matrix and for any correlation matrix $R$, there exists $H \in Chol(p)$ such that $R = H^T H$, as proved in [4]. Moreover, since $R$ is a positive definite matrix, there is a unique Cholesky factorization such that $R = H^T H$, thus defining a bijection between the two spaces. Therefore the idea is to map the sample of correlation matrices to the Cholesky manifold, use the geometry of the manifold to carry out the analysis, and then map back the predicted field to the space of correlation matrices.

In order to analyse the geometry of $Chol(p)$, let us first recall some properties of
the hypersphere \( S^q = \{ z \in \mathbb{R}^q : \| z \|_{\mathbb{R}^q} = 1 \} \). It can be equipped with a Riemannian manifold structure, by considering it as a Riemannian submanifold of the embedding Euclidean space \( \mathbb{R}^q \) with the usual inner product \( \langle x, y \rangle_{\mathbb{R}^q} = x^T y \), \( x, y \in \mathbb{R}^q \). The tangent space to \( S^q \) in \( z \) is \( T_z S^q = \{ y \in \mathbb{R}^q : \langle z, y \rangle_{\mathbb{R}^q} = 0 \} \), i.e. the set of vectors orthogonal to \( z \), which can be equipped with the usual inner product in \( \mathbb{R}^q \).

The exponential map, namely the function which projects an element \( y \) of \( T_z S^q \) to \( S^q \), and its inverse, the logarithmic map, are defined as follows:

\[
\exp_{z_0} (y) = \cos (\| y \|_{\mathbb{R}^q}) z_0 + \sin (\| y \|_{\mathbb{R}^q}) \frac{y}{\| y \|_{\mathbb{R}^q}} \\
\log_{z_0} (z) = \frac{d_{S^q} (z_0, z)}{\| P_{z_0} (z - z_0) \|_{\mathbb{R}^q}} P_{z_0} (z - z_0)
\]

where \( P_{z_0} (x) = x - \langle z_0, x \rangle_{\mathbb{R}^q} \) is the projection of an element \( x \in \mathbb{R}^q \) into the tangent space \( T_{z_0} S^q \) and \( d_{S^q} (z_0, z) = \arccos (\langle z_0, z \rangle_{\mathbb{R}^q}) \) is the great circle distance between two elements \( z_0 \) and \( z \) in \( S^q \).

These definitions can be extended to the Cholesky manifold by acting individually on each vector \( H^{(i)} \), for \( i = 2 \ldots p \).

Let us define a map \( \mathcal{R} : \mathbb{R} \otimes \bigotimes_{d=2}^p S^d \rightarrow \text{Chol} \) such that:

\[
\mathcal{R} \left( H^{(1)}, H^{(2)}, \ldots, H^{(p)} \right) = H = \begin{cases} 
H_{11} = H^{(1)} \\
H_{ij} = 0 & \text{if } i > j \\
H_{ij} = H^{(i)} & \text{otherwise}
\end{cases}
\]

The tangent space in a point \( H \in \text{Chol} \) is \( T_H \text{Chol} = \bigotimes_{q=2}^p T_{H^{(q)}} S^q \), i.e. if \( X \in T_H \text{Chol} \), then \( X \) is an upper triangular matrix \( p \times p \) such that \( X_{11} = 0 \) and \( X^{(q)} \in T_{H^{(q)}} S^q \) for \( q = 2 \ldots p \), where \( X^{(q)} \) again denotes the vector of the first \( q \) elements of the \( q \)-th column of \( X \). In order to project matrices from \( T_H \text{Chol} \) to \( \text{Chol} \), and viceversa, exponential and logarithmic maps are defined as follows:

\[
- \exp_H (X) = \mathcal{R} \left( 1, \exp_{H^{(2)}} (X^{(2)}), \ldots, \exp_{H^{(p)}} (X^{(p)}) \right), \\
- \log_H (Z) = \mathcal{R} \left( 0, \log_{H^{(2)}} (Z^{(2)}), \ldots, \log_{H^{(p)}} (Z^{(p)}) \right)
\]

where \( \exp_{H^{(q)}} (X^{(q)}) \) and \( \log_{H^{(q)}} (Z^{(q)}) \) denotes, respectively, the exponential and logarithmic map in \( S^q \), \( T_{H^{(q)}} S^q \), being \( H^{(q)} \), \( Z^{(q)} \) the vectors of the first \( q \) elements of the \( q \)-th column of \( H \) and \( Z \). The metric on the manifold is defined as

\[
d_{\text{Chol}(p)} (H, Z) = \sqrt{\sum_{q=2}^p d_{S^q}^2 (H^{(q)}, Z^{(q)})}
\]
1.3.1 Model and Kriging prediction

The reasoning is quite similar to the case of positive definite symmetric matrices, with just some slight differences. First of all, random correlation matrix $R_s$ at location $s \in D$ is decomposed in the random Cholesky factors $R_s = \chi_s^T \chi_s$, $\chi_s \in Chol(p)$. Then each factor is modelled as

$$\chi_s (\Sigma) = \exp_{\Sigma} \{ A (x(s); \beta) + \Delta (s) \}$$  \hspace{1cm} (1.7)

where $\exp_{\Sigma} (\cdot)$ denotes the exponential map on the Cholesky manifold in $\Sigma$ and $\Delta (s) \in T_\Sigma Chol(p)$. $A(x, \beta) \in T_\Sigma Chol(p)$ is described by a linear model, as in (1.3). The second difference appears in the estimation of the tangent point of the manifold. Indeed for the case of the (hyper)sphere – and, as a consequence, of the Cholesky manifold – unicity of the intrinsic mean is not guaranteed. For this reason, in the setting of correlation matrices, it is possible to use the sample extrinsic mean as tangent point $\Sigma$. This is obtained by computing the arithmetic mean for the column vectors and then projecting them on the sphere, i.e. rescaling each column vector to have norm 1.

The residuals’ semivariogram and the regression coefficients $\beta = (\beta_0, \ldots, \beta_r)$, with $\beta_{.,k} \in T_\Sigma Chol(p)$, can be estimated as in the case of positive definite symmetric matrices, using (1.2) and (1.4). Then, kriging can be performed, using (1.5) to get the prediction in $Chol(p)$ and then transforming it back into a correlation matrix.

2 Code structure

2.1 Overview

The main goal of the project is to provide the user with two functions: one that, given a collection of positive definite symmetric matrices and some other parameters, builds the model and another one which, given the model and some new locations, performs kriging prediction. From an implementative point of view, this translates to the following algorithm:
**Initialization**

If not given in input, compute \( \hat{\Sigma} \) as the intrinsic mean of the data. Then initialize \( \hat{\beta}^{(0)} \) using formula 1.4, with \( \hat{F} = I \).

**Main loop**

while (! converged )

- Compute the tangent space residuals \( \hat{\Delta}^{(m)}(s_i) = \log_{\hat{\Sigma}}(S_i) - A(x(s_i); \hat{\beta}^{(m-1)}) \)
- Estimate the empirical semivariogram \( \hat{\gamma}^{(m)}(h) \) from \( \hat{\Delta}^{(m)}(s_1), \ldots, (s_n) \)
- Fit the model semivariogram \( \hat{\gamma}^{(m)}(h) \) via weighted least square and compute the corresponding covariogram, \( \hat{C}^{(m)} \), and covariance matrix \( \hat{F}^{(m)}_{ij} = \hat{C}^{(m)}(\|s_i - s_j\|) \)
- Obtain a new estimate for the \( \beta \)'s as \( \hat{\beta}^{(m)}_{lq} = \left( Z^T \hat{F}^{-1} Z \right)^{-1} Z^T \hat{F}^{-1} Y_{lq} \left( \hat{\Sigma} \right) \)

end

**Kriging**

Once the model is estimated, given an unobserved location \( s_0 \), the prediction is given by
\[ \hat{S}_0 = \exp_{\hat{\Sigma}}(\hat{\beta}_{.0}(\hat{\Sigma}) + \sum_{k=1}^r \hat{\beta}_{.k}(\hat{\Sigma})x_k(s_0) + \sum_{i=1}^n \lambda_{0i} \hat{\Delta}(s_i)) \]

At first, taking the cue from some code provided us by Professor Menafoglio, we developed a stable version of the algorithm in R, available at [https://github.com/LucaTorriani/KrigingManifoldData/tree/PureRC ode](https://github.com/LucaTorriani/KrigingManifoldData/tree/PureRC ode). Then, we implemented the following classes and functions to translate the entire algorithm in C++ (available at [https://github.com/LucaTorriani/KrigingManifoldData](https://github.com/LucaTorriani/KrigingManifoldData)):

```cpp
class Coordinates;

namespace design_matrix {
  class DesignMatrix;
  class InterceptDM : public DesignMatrix;
  class Coord1DM : public DesignMatrix;
  class Coord2DM : public DesignMatrix;
  class AdditiveDM : public DesignMatrix;
};

namespace distances {
  class Distance;
  class EuclDist : public Distance;
  class GeoDist : public Distance;
};

namespace distances_manifold {
  class DistanceManifold;
};
```
class Frobenius : public DistanceManifold;
class LogEuclidean : public DistanceManifold;
class SqRoot : public DistanceManifold;
class Chol : public DistanceManifold;
};

namespace distances_tplane{
    class DistanceTplane;
    class Frobenius : public DistanceTplane;
    class FrobeniusScaled : public DistanceTplane;
    class Chol : public DistanceTplane;
};

namespace variogram_evaluation{
    class EmpiricalVariogram;
};

namespace generic_factory{
    template <
        typename AbstractProduct ,
        typename Identifier ,
        typename Builder = function < unique_ptr < AbstractProduct > () >
    >
    class Factory;
};

namespace generic_factory {
    template < typename Factory , typename ConcreteProduct >
    class Proxy;
};

namespace variogram_evaluation {
    class FittedVariogram;
    class GaussVariogram : public FittedVariogram;
    class ExpVariogram : public FittedVariogram;
    class SphVariogram : public FittedVariogram;
};

namespace map_functions {
    class logarithmicMap;
    class logMapProb : public logarithmicMap;
    class logMapLogEucl : public logarithmicMap;
    class logMapSqRoot : public logarithmicMap;
    class logMapChol : public logarithmicMap;
}
class exponentialMap;
class expMapFrob : public exponentialMap;
class expMapLogEucl : public exponentialMap;
class expMapSqrtRoot : public exponentialMap;
class expMapChol : public exponentialMap;
);

namespace model_fit {
    class Model;
};

namespace matrix_manipulation {
    MatrixXd expMat(const MatrixXd &);
    MatrixXd logMat(const MatrixXd &);
    MatrixXd sqrtMat(const MatrixXd &);

    vector<MatrixXd> bigMatrix2VecMatrices(const MatrixXd &,
        unsigned int, const string &);
    MatrixXd VecMatrices2bigMatrix(const vector<MatrixXd> &);
    MatrixXd Chol_semidef(const MatrixXd &);
    MatrixXd Chol_decomposition(const MatrixXd &);
};

MatrixXd intrinsic_mean_C(const vector<MatrixXd> &,
    string, map_functions::logarithmicMap &,
    map_functions::exponentialMap &,
    distances_tplane::DistanceTplane &, double, const Vec &,
    const Vec &);

MatrixXd extrinsic_mean(const vector<MatrixXd> &, const Vec &);

// Interface functions
RcppExport SEXP get_model(SEXP s_data_manifold,
    SEXP s_coordinates, SEXP s_X, SEXP s_Sigma, SEXP s_distance,
    SEXP s_manifold_metric, SEXP s_ts_metric, SEXP s_ts_model,
    SEXP s_vario_model, SEXP s_n_h, SEXP s_max_it,
    SEXP s_tolerance, SEXP s_max_sill, SEXP s_max_a,
    SEXP s_weight_vario, SEXP s_distance_matrix_tot,
    SEXP s_data_manifold_tot, SEXP s_coordinates_tot,
    SEXP s_X_tot, SEXP s_hmax, SEXP s_indexes_model,
    SEXP s_weight_intrinsic, SEXP s_tolerance_intrinsic,
    SEXP s_weight_extrinsic, SEXP s_suppressMes,
2.2 Factories to handle run-time parameters

In the algorithm there are many input parameters that can be fixed run-time. In particular a user chooses the type of the fitted variogram, the geographic distance, the model on the tangent space and both the distance on the tangent space and on the manifold, which determines the associated maps. To handle all these parameters run-time, we implemented template factories, and a proxy to register them. In the following we describe in details, as an example of the factory structure, the classes that manage the choice of the manifold distance. Similarly, all the other run-time choices are handled.

```c++
#ifndef _DISTANCE_MANIFOLD_HPP_

SEXP s_tolerance_map_cor);

RcppExport SEXP get_kriging (SEXP s_coordinates,
    SEXP s_new_coordinates, SEXP s_Sigma, SEXP s_distance,
    SEXP s_manifold_metric, SEXP s_ts_model, SEXP s_vario_model,
    SEXP s_beta, SEXP s_gamma_matrix, SEXP s_vario_parameters,
    SEXP s_residuals, SEXP s_X_new, SEXP s_tolerance_map_cor);

RcppExport SEXP get_model_and_kriging (SEXP s_data_manifold,
    SEXP s_coordinates, SEXP s_X, SEXP s_Sigma, SEXP s_distance,
    SEXP s_manifold_metric, SEXP s_ts_metric, SEXP s_ts_model,
    SEXP s_vario_model, SEXP s_n_h, SEXP s_max_it,
    SEXP s_tolerance, SEXP s_max_sill, SEXP s_max_a,
    SEXP s_weight_vario, SEXP s_distance_matrix_tot,
    SEXP s_data_manifold_tot, SEXP s_distance_matrix_tot,
    SEXP s_hmax, SEXP s_indexes_model, SEXP s_weight_intrinsic,
    SEXP s_tolerance_intrinsic, SEXP s_weight_extrinsic,
    SEXP s_new_coordinates, SEXP s_X_new, SEXP s_suppressMes,
    SEXP s_tolerance_map_cor);

RcppExport SEXP intrinsic_mean (SEXP s_data, SEXP s_N,
    SEXP s_manifold_metric, SEXP s_ts_metric, SEXP s_tolerance,
    SEXP s_weight_intrinsic, SEXP s_weight_extrinsic,
    SEXP s_tolerance_map_cor);

RcppExport SEXP distance_manifold (SEXP s_data1, SEXP s_data2,
    SEXP s_N1, SEXP s_N2, SEXP s_manifold_metric)

#endif
```
# define _DISTANCE_MANIFOLD_HPP_

#include "Helpers.hpp"
#include <vector>
#include <utility>
#include <map>
#include <memory>

using namespace Eigen;
namespace distances_manifold{
  class DistanceManifold{
    public:
      virtual double compute_distance(const MatrixXd&, const MatrixXd&) const = 0;
      virtual ~DistanceManifold() = default;
    };

    class Frobenius : public DistanceManifold{
      public:
        double compute_distance(const MatrixXd&, const MatrixXd&) const override;
        ~Frobenius() = default;
    };

    class LogEuclidean : public DistanceManifold{
      public:
        double compute_distance(const MatrixXd&, const MatrixXd&) const override;
        ~LogEuclidean() = default;
    };

    class SqRoot : public DistanceManifold{
      public:
        double compute_distance(const MatrixXd&, const MatrixXd&) const override;
        ~SqRoot() = default;
    };

    class Chol : public DistanceManifold{
      public:
        double compute_distance(const MatrixXd&, const MatrixXd&) const override;
        ~Chol() = default;
    };
}
In particular, a user can choose among four different types of manifold distances: Frobenius, FrobeniusScaled, SqRoot and Cholesky (the last one handles correlation matrices). Thus, we implemented a pure virtual class DistanceManifold and four children classes which override the member function compute_distance according to the distance chosen. In Listing 2.2 it is shown how the factory is used, taking into account the typedefs in Listing 2.3, and the instantiations in Listing 2.4.

```cpp
{
// ... some code
std::string distance_Manifold_name = Rcpp::as<std::string>(
    s_manifold_metric);
manifold_factory::ManifoldFactory& manifold_fac (manifold_factory::ManifoldFactory::Instance());
std::unique_ptr<distances_manifold::DistanceManifold>
    theManifoldDist = manifold_fac.create(distance_Manifold_name);
// ... some other code
}
```

Listing 2.2: HelpersFactory.hpp

```cpp
#ifndef _HELPERS_FACTORY_HPP_
define _HELPERS_FACTORY_HPP_

// ... Other includes
#include "DistanceManifold.hpp"
#include "DistanceTplane.hpp"

// ... Other code
namespace manifold_factory{
    typedef generic_factory::Factory<
        distances_manifold::DistanceManifold, std::string
    > ManifoldFactory;

template<typename ConcreteProduct>
using ManifoldProxy = generic_factory::Proxy<
    ManifoldFactory, ConcreteProduct
>;
```

14
### Listing 2.3: Choice of the manifold distance

```cpp
#include "HelpersFactory.hpp"

namespace {
    // ... Other using directives
    using manifold_factory::ManifoldProxy;

    ManifoldProxy<distances_manifold::Frobenius>
    manfrob("Frobenius");
    ManifoldProxy<distances_manifold::LogEuclidean>
    manlogeucl("LogEuclidean");
    ManifoldProxy<distances_manifold::SqRoot>
    mansqroot("SquareRoot");
    ManifoldProxy<distances_manifold::Chol>
    manchol("Correlation");
    // ... Other instantiations
}
```

### Listing 2.4: Registration of the factory through the proxy

2.3 Gauss-Newton to compute variogram parameters

Given the empirical variogram, the problem of computing the parameters $\theta$ of the fitted variogram consists in determining the optimal values minimizing the sum of square of the residuals. Namely,

$$
\min_{\theta \in \mathbb{R}^3} \frac{1}{2} \sum_{i=1}^{n} r_i^2
$$

where $r_i(\theta) = \gamma_m(h_i, \theta) - \hat{\gamma}(h_i)$ and $\mathbf{r} = (r_1, \ldots, r_n)^T$. Defining the Jacobian of the residuals $J(\theta) = \left[ \frac{\partial r_i}{\partial \theta_j} \right] = \nabla r_i(\theta)$, we can write

$$
\nabla f(\theta) = J(\theta)^T \mathbf{r}(\theta)
$$

$$
H_f(\theta) = J(\theta)^T J(\theta) + \sum_{i=1}^{n} \nabla^2 r_i(\theta) r_i(\theta)
$$
Now we consider Gauss-Newton method such that, in the updating rule \( \theta^{(k+1)} = \theta^{(k)} + \alpha^{(k)} \rho^{(k)} \), the direction \( \rho^{(k)} \) solves \( J^{(k)T} J^{(k)} \rho^{(k)} = -J^{(k)T} r^{(k)} \) instead of \( H_f^{(k)} \rho^{(k)} = -\nabla f^{(k)} \). In particular, \( \rho^{(k)} \) is the least square solution of

\[
\min_{\rho \in \mathbb{R}^3} \frac{1}{2} \| J^{(k)} \rho + r^{(k)} \|^2 \quad (2.2)
\]

where \( J^{(k)} = J(\theta^{(k)}) \), \( r^{(k)} = r(\theta^{(k)}) \), \( H_f^{(k)} = H_f(\theta^{(k)}) \) and \( \nabla f^{(k)} = \nabla(\theta^{(k)}) \).

In order to determine \( \alpha_k \) at each iteration, we used the backtrack algorithm described in [6], which ensures that the \( \alpha_k \) computed satisfies the Wolfe conditions:

```
set \( \alpha = 1 \)
set \( c = 10^{-4} \)
set \( \sigma = 0.25 \)
while \( f(x^{(k)} + \alpha d^{(k)}) > f(x^{(k)}) + c \alpha d^{(k)} \nabla f(x^{(k)}) \)
  \( \alpha = \alpha \sigma \)
end
set \( \alpha_k = \alpha \)
```

In our code \( \theta \in \mathbb{R}^3 \) represents the vector \((\text{nugget, sill} - \text{nugget, practical range})\) and the starting point \( \theta^{(0)} \) is computed by the member function \text{get\_init\_par} of the class \text{FittedVariogram}, overridden by the three children class, \text{GaussVariogram}, \text{ExpVariogram} and \text{SphVario}. As far as the nugget is concerned it is always initialized as a weighted median of the first two values of the empirical variogram. The starting values for sill and practical range, instead, are defined in slightly different ways according to the model variogram, however all the three member functions initialize the former with a function of the weighted median of the last four values of the empirical variogram, and the latter with a function of \( h_{\text{max}} \) (the maximum distance for which the variogram is computed). The minimization is implemented by means of the two functions \text{backtrack} and \text{evaluate\_par\_fitted} (members of \text{FittedVariogram}). It is worth noting that there are actually two versions of the second function, namely \text{evaluate\_par\_fitted\_E} and \text{evaluate\_par\_fitted\_W}, which differ only in the stopping criterion. The former is based on the decrease of the norm of the variogram residuals (\text{variogram residuals} = \text{fitted variogram values} - \text{empirical variogram values}), while in the latter the difference between these decreases at iteration \( k \) and \( k+1 \) is considered. Since for statistical reasons only some values of the parameters are reasonable, we had to add some box constraints to the minimization. This problem becomes extremely evident and compelling for non stationary fields. To solve it we introduce some \text{barrier values} that can’t be crossed by the parameters. When, according to the updating rule a parameter would cross
one of these boundaries, we fix it to the very barrier value. Namely, the constraints are:

\[
0 \leq \text{nugget} \\
\text{nugget} \leq \text{sill} \leq \text{max_sill} \\
0 \leq \text{practical range} \leq \text{max_a}
\]

The values for max_sill and max_a can be provided by the user. Otherwise they are computed as follows:

\[
\text{max_sill} = 1.15 \times \max(\text{emp_vario_values}) \\
\text{max_a} = 1.15 \times h_{\text{max}}
\]

2.4 Some technical remarks

- In the code the library \texttt{Eigen} is used to handle matrices and vectors.

- In the function \texttt{update_model} of the class \texttt{Model} the $\beta$'s are computed via the closed form shown in formula 1.4.

- For covariance data we considered LDLT decomposition to invert matrices. For correlation data, instead, we used LLT decomposition on positive definite matrices, while, due to numerical issues, on matrices only positive semidefinite we applied an algorithm provided by Professor Davide Pigoli, reported in Listing 2.5.

- Given two points, with their longitudes and latitudes, their great-circle distance is computed via the Haversine formula (see [11]).

- The design matrix, the matrix of the coordinates, the distance matrix and the data on the tangent space are saved using shared pointers, since they are constant, used by multiple classes and possibly quite large.

- In the implementation of the kriging we used two lambda functions to make the code more readable.

```cpp
MatrixXd matrix_manipulation::Chol_semidef (const MatrixXd & M1) {
    unsigned int p(M1.rows());
    MatrixXd result(p,p);
    result.setZero(p,p);
    result(0,0)=1;
    double tmp;
}
```
for (size_t i=1; i<p; i++) {
    for (size_t j=0; j<i; j++) {
        if (j==0) result(i,j) = M1(i,j);
        else {
            result(i,j) = M1(i,j) -
            ((result.row(i)*result.row(j).transpose()).value())/result(j,j)
        }
    }
    tmp=result.row(i).norm();
    result(i,i) = sqrt(1-tmp*tmp);
}
return(result.transpose());

Listing 2.5: Algorithm for Cholesky decomposition

2.5 Code documentation

The Doxygen documentation of the C++ code can be built from the src folder using the provided Doxyfile. To run Doxygen type:

doxxygen Doxyfile

In order to create a new documentation with different settings, the user needs to create a new configuration file, i.e. Doxyfile, with:

doxxygen -g <config-file>

that can be edited to customize the output. To get the Reference Manual, it is enough to go to the latex subfolder that has been generated and type make.

A copy of the manual obtained with the Doxyfile in src is available at https://github.com/LucaTorriani/KrigingManifoldData/blob/master/Documentation/Doxygen_documentation.pdf.

3 Interface R/C++

The ‘Writing R Extensions’ manual (see [7]) describes in detail how to augment R with compiled code. The R application programming interface (API) described there is based on a set of functions and macros operating on SEXP (pointers to SEXPREC or ‘S expression’ structures) which are the internal representation of
R objects. The \texttt{Rcpp} package (see [8]), simplifies the embedding of C++ code in R. Indeed the function \texttt{.Call} provides a powerful, yet quite simple, way to use precompiled code in R. Moreover the \texttt{RcppEigen} package, allows integration between R and the \texttt{Eigen} library.

The most effective way to make code easily accessible to R users is to create an R package whose functions exploit C++ scripts to improve the computational speed. In order to achieve this, the code must be structured as follows:

- **src** directory: containing all the source C++ files, along with the \texttt{Makevars} file. In particular it contains one or more \texttt{.cpp} files with the implementation of the functions that will be invoked by the R code. Those, that we will call \textit{interface functions} from now on, receive SEXP parameters (that will be explicitly cast using \texttt{Rcpp::as}), and return objects compatible with R exploiting the \texttt{Rcpp::wrap} function.

- **R** directory: containing all the R functions, included the ones calling the \textit{interface functions}. Those reporting the \texttt{@export} directive constitute the functions provided to the users by the package.

- **man** directory: containing documentation file (\texttt{.Rd}) for the R functions. It is automatically generated using the function \texttt{roxygenise} of the R package \texttt{Roxygen2}.

- **data** directory: containing \texttt{.RData} files used in the examples.

- **DESCRIPTION** file: containing basic information about the package, such as its name, the authors and the licence. It also provides a brief description of the package and information useful for the installation.

- **NAMESPACE** file: containing namespace directives to manage the function exports to the R environment. Like the **man** directory it is automatically generated using \texttt{Roxygen2}.

### 3.1 Debug

A drawback of augmenting R code with compiled code is that it may happen that R session crashes due to some memory leakage. Unfortunately, the R session does not provide any explanation of the fault, thus it becomes very difficult to find and fix the problem. A very useful tool to detect this type of problem is Valgrind, which can be easily used executing this command from the terminal
Valgrind helped us to detect memory leakages that occurred every time an object created through the factories went out of scope. To understand the issue, let us consider, for instance, the classes to manage the type of distance on the tangent space:

```cpp
#include "Helpers.hpp"
#include <vector>
#include <utility>
#include <map>
#include <functional>
#include <memory>

namespace distances_tplane {

class DistanceTplane {
public:
    double compute_distance(const MatrixXd & M1,  
                            const MatrixXd & M2) const;  
    virtual double norm(const MatrixXd & M1) const = 0;  
    virtual void set_members(const MatrixXd & Sigma) = 0;  
    virtual ~DistanceTplane() = default;
};

class Frobenius : public DistanceTplane{
public:
    ~Frobenius() = default;
    double norm(const MatrixXd & M1) const override;  
    void set_members(const MatrixXd & Sigma) override;
};

class FrobeniusScaled : public DistanceTplane{
    MatrixXd _SigmaInv;
    unsigned int _p;
public:
    ~FrobeniusScaled() = default;
    double norm(const MatrixXd & M1) const override;  
    void set_members(const MatrixXd & Sigma) override;
};
```
Our problem lay in the fact that, at first, we hadn’t defined the destructor of the \texttt{DistanceTplane} class as pure virtual. Therefore the children classes used the father’s destructor and when, for example, an object of class \texttt{FrobeniusScaled} was deleted, the space of the two private members was not freed.

## 4 Manifoldgstat package

The R package developed through this project is called \texttt{Manifoldgstat} and it provides the users with the following functions:

- \texttt{model\_GLS} that, given the coordinates and corresponding manifold values, creates a GLS model on the tangent space.

- \texttt{kriging} that, given the GLS model, performs kriging prediction on new locations.

- \texttt{model\_kriging} that, given the coordinates and corresponding manifold values, firstly creates a GLS model on the tangent space, and then performs kriging on the new locations.

- \texttt{distance\_manifold} that computes the manifold distance between symmetric positive definite matrices.

- \texttt{intrinsic\_mean} that computes the intrinsic mean of a given set of symmetric positive definite matrices.
A detailed description of these functions can be found in the package documentation, available at https://github.com/LucaTorriani/KrigingManifoldData/blob/master/Documentation/Manifoldgstat_documentation.pdf.

5 Results

We tested our code on a simulated stationary field over a rectangular domain, both in case of covariance and correlation matrices.

5.1 Positive definite symmetric matrices

In this first test the data consisted in 1000 positive definite symmetric matrices, represented in fig. 5.1 (for the sake of clarity one ellipse every two is drawn).

![Figure 5.1: Stationary random field of covariance matrices](image)

From this data we extracted a sample of 250 matrices, in fig. 5.2, and used it to fit the model. The variogram obtained is represented in fig. 5.3. Then, once the
model was computed, through kriging we tried to predict the values of the field in all the 1000 locations.

```r
data_manifold <- Manifoldgstat::rCov
coords_model <- Manifoldgstat::rGrid # Locations of the data
coords_tot <- Manifoldgstat::gridCov # Locations where to predict
Sigma <- matrix(c(2,1,1,1), 2,2)

model <- Manifoldgstat::model_GLS(
  data_manifold = data_manifold, coords = coords_model,
  Sigma = Sigma, metric_manifold = "Frobenius",
  metric_ts = "Frobenius", model_ts = "Coord1",
  vario_model = "Spherical", distance = "Eucldist",
  max_it = 100, tolerance = 1e-7, plot = TRUE)

result_tot <- kriging(GLS_model = model, coords = coords_model,
  new_coords = coords_tot, model_ts="Coord1",
  vario_model= "Spherical", metric_manifold = "Frobenius",
  distance="Eucldist")
```

Listing 5.1: CodeSD.R

![Empirical and fitted variogram](image)

Figure 5.3: Empirical and fitted variogram

In fig. 5.4 the red ellipses represent the estimated covariance matrices at the 250 locations corresponing to the data used to build the model, while the blue ones
are those regarding the other locations. The prediction errors, computed, in every location, as the Frobenius norm of the difference between the real value and the predicted one, are depicted in fig 5.5. As we would expect the error in correspondence of the 250 locations used to create the model is zero, since kriging prediction interpolates the data. Eventually, we compared the performance of the C++ code w.r.t. the original R code version using the package benchr and the results are reported in fig 5.6. In the last column it is shown that the C++ version is about 1410 times faster than the corresponding R version.

![Figure 5.4: Predicted values](image1)

![Figure 5.5: Errors of the kriging prediction](image2)

![Figure 5.6: Output function benchmark of the R package benchr](image3)

### 5.2 Correlation matrices

In this second example we used the same field of section 5.1, but rescaled in order to have 1000 correlation matrices. (For the sake of clarity one ellipse every two is drawn
in fig. 5.7). As before, from this field we extracted a sample of 250 matrices (shown in fig. 5.2), fitted the model and eventually tried to predict, through kriging, the values of the field in all the 1000 locations. The variogram obtained is represented in fig. 5.9.

Figure 5.7: Stationary random field of correlation matrices

Figure 5.8: Sample of the stationary field of correlation matrices
In fig. 5.10, the estimated correlation matrices at the new locations are depicted in blue, while the predictions for the data used to build the model are represented in red. Figure fig 5.11, instead, shows the prediction errors. Finally, from fig 5.12 we can conclude that in this case the C++ code is about 1740 times faster than the corresponding R version.
6 Conclusion

Starting this project we aimed at implementing a faster version of our R algorithm, to be then able to run a lot of different simulations for statistical purposes. As demonstrated in 5.1 and 5.2 the C++ code is definitely faster than the R code, thus the main objective was accomplished. Moreover the use of factories (and the proxy for the registration), makes future development of the code easier. It is worth noting that this scheme also helped us during the work. For example, the possibility of handling correlation matrices, besides covariace matrices, was easily added when the code was almost ended. We just had to implement the maps and the distances relative to the correlation matrices and add them to the existing factories.

Since the study of symmetric positive definite matrices (SPD) intersects several research filed, our code could be readapted for a lot of different applications. We provide some examples suggested by [9], one of the numerous articles that through differential geometry develop methods for the manifold of SPD matrices. In computational anatomy, an SPD deformation vector is computed to capture the directional information of shape change at each location in an image. In diffusion tensor imaging, a $3 \times 3$ SPD diffusion tensor, which tracks the effective diffusion of water molecules, is estimated at each voxel (a three-dimensional pixel) of an imaging space. In functional magnetic resonance imaging, an SPD covariance matrix is calculated to delineate functional connectivity between different neural assemblies involved in achieving a complex cognitive task or perceptual process.
6.1 Future work

The two techniques described in 1.2.1 and 1.3.1 for positive definite symmetric matrices and correlation matrices, respectively, are based on the stationarity of the process generating the data. However, in many application, observed data are not compatible with a global stationarity assumption. A possible solution to overcome this problem consists in integrating the analysis in a more complex algorithm, called Random Domain Decomposition (see [10]). The basic idea is to use simple, local and repeated analyses instead of an unique global and complex one, through a *divide et impera* strategy. During the *divide* step, the spatial domain is randomly partitioned into a set of disjoint sub-regions, within which local geostatistical analyses are performed. These local and weak analyses are repeated for different realizations of the random domain decomposition and then aggregated into a final strong analysis during the *impera* step. In other words, in each single cell of the domain decomposition, where stationarity is assumed to hold true, an empirical variogram is computed, a model variogram is fitted and eventually the $\beta$ parameters are estimated.

Our code has already been arranged for dealing with the additional computation required by its embedding in the RDD algorithm. For instance it provides the possibility of considering a kernel-weighted estimation of the local variogram, to cope with the bias-variance trade-off introduced by the procedure.

As said before another field of application could be diffusion tensor imaging. In particular, for the Applied Statistics project we worked on brain imaging, where diffusion tensors of water are used to identify the position of neural fibers. It may be interesting to apply our algorithm to these data in order to look for significant difference between the measured diffusion tensor and the predicted ones, which could suggest the presence of a tumor altering the distribution of the axons.

7 Installation

There are three different methods to install the package. If one of the last two is used, the package devtools has already to be installed.

- Download source code from the github repository, unzip the file and run from the terminal:

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

- Download source code from the github repository, unzip the file and run from the package root folder the following commands to create the package and to build the documentation in Roxygen:

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

- Install the package directly from R:

  ```
  library(devtools)
  install_github("LucaTorriani/KrigingManifoldData")
  ```

To test the successful installation of the package, it is possible to run the examples directly from R with the following commands:

```
example(model_GLS, "Manifoldgstat")
example(kriging, "Manifoldgstat")
example(model_kriging, "Manifoldgstat")
example(distance_manifold, "Manifoldgstat")
example(intrinsic_mean, "Manifoldgstat")
```
References


