POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Package 'LocallyStationaryModels'

APSC course project - 8 cfu
Prof. Luca Formaggia

Authors: **Luca Crippa, Giacomo De Carlo**

Students' IDs: 10614479 (Luca Crippa), 10609260 (Giacomo De Carlo)
Advisor: Prof. Alessandra Menafoglio
Co-advisors: Dott. Riccardo Scimone
Academic Year: 2020-21

Contents

Abstract

In this report we describe our implementation of an R package based on source C++ code, of the model proposed in (1) for covariance estimation of non-stationary spatial random processes. We first introduce general concepts about spatial data and geostatistics, assumptions of stationarity of a random process and simple methods for covariance estimation for such processes, basing our introduction on lecture notes by Professor Menafoglio (4); for a more detailed discussion on geostatistics and spatial data we can refer to (8). We then describe the method in (1) and how we structured the implementation, since no code was provided by the authors. The following sections regards the structure of the code: how the algorithms and the interface between R and C++ were implemented. Eventually we propose some examples of the method tested on real and simulated data, a brief tutorial of the use and functions of the R package and a brief guide of installation.

Chapter 1

Fundamentals of Geostatistics

1.1 Geostatistical Data

Let $\mathbf{s} \in \mathbb{R}^d$ be a generic data location in d-dimensional Euclidean space and suppose that the potential datum $\mathbf{Z_s}$ at spatial location $\mathbf{s}$ is a random quantity. Now let $\mathbf{s}$ vary over index set $D \subset \mathbb{R}^d$ so as to generate the multivariate random field (or random process)

$$\{\mathbf{Z_s} : \mathbf{s} \in D\}$$

In the setting of geostatistical data, D is a fixed domain in $\mathbb{R}^d$ that contains a d-dimensional rectangle with non-null volume. The stochastic process is indexed by the continuous spatial variable $\mathbf{s}$ in D and $\mathbf{Z_s}$ is a random variable (or vector) observed at the location $\mathbf{s}$ in D.

1.2 Isotropic and stationary random fields

For prediction purposes further statistical assumptions need to be made. A very useful statistical property for a random process is stationarity. Various degrees of stationarity can be defined; we will use the common decomposition of process $\mathbf{Z_s}$ as:

$$\mathbf{Z_s} = \mathbf{m_s} + \boldsymbol{\delta}_\mathbf{s}\,,\ \mathbf{s} \in D$$

Definition 1.2.1 *A spatial process $\mathbf{Z_s}$ with spatial domain $D \subset \mathbb{R}^d$ is said to be strongly stationary if for every finite family of locations $\{\mathbf{s}_k\}_{k=1}^n$ in D and for every $\mathbf{h} \in \mathbb{R}^d$ the random vectors $(\mathbf{Z_{s_1}}, ..., \mathbf{Z_{s_N}})$ and $(\mathbf{Z_{s_1+h}}, ..., \mathbf{Z_{s_N+h}})$ have the same joint law.*

The property of strong stationarity implies a weaker form of stationarity, easier to be verified in real applications.

Definition 1.2.2 *A spatial process $\mathbf{Z_s}$ with spatial domain $D \subset \mathbb{R}^d$ is said to be second-order stationary if the following conditions hold*

- $\mathbb{E}[Z_s] = m$, *for all* $\mathbf{s}$ *in* D;

- $Cov(Z_{\mathbf{s}_i}, Z_{\mathbf{s}_j}) = \mathbb{E}[(Z_{\mathbf{s}_i} - m)(Z_{\mathbf{s}_j} - m)] = C(\mathbf{h})$, *for all* $\mathbf{s}_i, \mathbf{s}_j$ *in* D, $\mathbf{h} = \mathbf{s}_i - \mathbf{s}_j$;

Function C is said *ovariogram.*
Strong stationarity always implies second-order stationarity. In general, the reverse implication does not hold, unless the process is Gaussian: in this case, the finite dimensional laws are completely characterized by the first two moments and definitions 1.2.1 and 1.2.2 coincide. An even weaker characterization of stationarity can be defined as follows:

Definition 1.2.3 *A spatial process* $\mathbf{Z_s}$ *with spatial domain* $D \subset \mathbb{R}^d$ *is said to be intrinsically stationary if the following conditions hold*

- $\mathbb{E}[Z_s] = m$, *for all* $\mathbf{s}$ *in* D;
- $Var(Z_{\mathbf{s}_i} - Z_{\mathbf{s}_j}) = \mathbb{E}[(Z_{\mathbf{s}_i} - Z_{\mathbf{s}_j})^2] = 2\gamma(\mathbf{h})$, *for all* $\mathbf{s}_i, \mathbf{s}_j$ *in* D, $\mathbf{h} = \mathbf{s}_i - \mathbf{s}_j$;

The function γ is commonly called semivariogram, and the function 2γ is the variogram.

Definition 1.2.4 *An intrinsic stationary process* $\mathbf{Z_s}$ *with spatial domain* $D \subset \mathbb{R}^d$ *is said isotropic if its variogram is isotropic, i.e.,*

$$Var(Z_{\mathbf{s}_i} - Z_{\mathbf{s}_j}) = 2\gamma(h), \;\; h = \|\mathbf{h}\| = \|\mathbf{s}_i - \mathbf{s}_j\|, \;\; \mathbf{s}_i, \mathbf{s}_j \in D$$

1.3 Valid variograms

A variogram fulfilling the following properties is said to be valid:

- Conditional negative definiteness:

$$\sum_i \sum_j \lambda_i \lambda_j \gamma(\mathbf{s}_i - \mathbf{s}_j) \leq 0, \; \forall \mathbf{s}_i, \mathbf{s}_j \in D, \; \forall \lambda_i, \lambda_j \text{ s.t. } \sum_i \lambda_i = 0$$

- Symmetry: $\gamma(\mathbf{h}) = \gamma(-\mathbf{h})$
- Non-negativity: $\gamma(\mathbf{h}) \geq 0$
- Zero at the origin: $\gamma(\mathbf{0}) = 0$
- Sub-quadratic growth: $\lim_{\|\mathbf{h}\| \to \infty} \frac{2\gamma(\mathbf{h})}{\|\mathbf{h}\|^2} = 0$

In this case, the semivariogram is related to the covariogram via the identity

$$\gamma(\mathbf{h}) = C(\mathbf{0}) - C(\mathbf{h}), \;\; \mathbf{h} \in \mathbb{R}^d$$

A list of valid variogram models implemented in our package is presented in Appendix A.

1.4 Variogram Estimation

The estimation of the variogram from data (i.e. a realization of a spatial random process $\mathbf{Z_s}$) is one of the most important problems in geostatistics.
The assumptions of isotropy and stationarity are very useful for this task. From definitions 1.2.3 and 1.2.4 we have:

$$2\gamma(h) = \mathbb{E}[(Z_{\mathbf{s}} - Z_{\mathbf{s}+\mathbf{h}})^2], \;\; h = \|\mathbf{h}\|$$

This strongly suggests that we can use an empirical mean as an estimator for the variogram function.
The usual procedure for variogram estimation and fitting consists in the following steps:

- Estimation of the empirical variogram. We set a spatial cutoff b, a threshold above which we think spatial autocorrelation is not meaningful. We divide the interval in a fixed number of K sub-intervals s.t. the $k-$th is defined as: $I_k = (\frac{(k-1)b}{K}, \frac{kb}{K})$ and

  $$N_k = \{(i,j) : \|\mathbf{s}_i - \mathbf{s}_j\| \in I_k\}$$

  n_k is the cardinality of such set and $h_k = \sum_{(i,j)\in N_k} \|\mathbf{s}_i - \mathbf{s}_j\|$
  We can now calculate the *binned semivariogram*

  $$\hat{\gamma}(h_k) = \frac{1}{2n_k} \sum_{(i,j)\in N_k} (Z_{\mathbf{s}_i} - Z_{\mathbf{s}_j})^2$$

  The total number K of sub-intervals is key to provide sensible estimates, as a trade-off exists between overfitting and oversmoothing: the cardinalities n_k need to be sufficient to guarantee the estimate precision, while the number K of classes is to be appropriate to catch the variogram structural properties,such as sill, range and behavior near the origin.

- Fitting of the semivariogram model. In order to estimate the covariance structure of the underlying random process a parametric model for a valid variogram, with parameter vector $\boldsymbol{\theta}$, is usually employed. $\gamma(h;\boldsymbol{\theta})$ is the value of the variogram model with specific parameter vector $\boldsymbol{\theta}$ at distance h.
  $\boldsymbol{\theta}$ is estimated by fitting a given minimization criterion, it can either be ordinary least squares (OLS), generalized least squares (GLS) or weighted least squares (WLS), which is the creterion used by our package.

  $$\hat{\boldsymbol{\theta}} = \underset{\theta\in\Theta}{\arg\min} \sum_{k=1}^{K} (\hat{\gamma}(h_k) - \gamma(h_k;\boldsymbol{\theta}))^2$$

  By estimating $\boldsymbol{\theta}$ we can retrieve the full structure of the covariance of the process.

Assumptions of isotropy and second-order stationarity greatly simplify the problem of covariance estimation, but in many cases this assumptions are too strict and

do not fit very well real use cases. The estimator for the binned variogram is unbiased (for $K \to \infty$) but only if we assume some degree of stationarity, in fact, in general we have:

$$\mathbb{E}[(Z_{\mathbf{s}} - Z_{\mathbf{s}+\mathbf{h}})^2] = 2\gamma(\|\mathbf{h}\|) + (m_{\mathbf{s}} - m_{\mathbf{s}+\mathbf{h}})^2$$

The following sections deal with the theoretical settings and practical implementation of weak form of nonstationarity and anisotropy, this framework and methods can be used to cover a much wider class of processes.

Chapter 2

Local Stationarity and Variogram Estimation

In this chapter we present the estimation method for non-stationary models proposed in (1) and our algorithmic implementation.

2.1 Local Stationarity

As we already mentioned, the assumptions of having a constant mean $m_{\mathbf{s}}$ and covariance function C across the whole domaing D of our spatial random process $\mathbf{Z_s}$ is often too strict, but by assuming that these functions vary "slowly" in our domain we can inlcude a much wider class of processes, hence the definition of *local stationarity*. (2)

A random process $\mathbf{Z_s}$ will be *locally stationary* or *quasi-stationary* if its mean function $m_{\mathbf{s}}$ and its covariance function $C(\cdot,\cdot)$ are such that:

- $m_{\mathbf{s}}$ is a very regular function varying slowly in space at the scale of the available information; more precisely, $m_{\mathbf{s}}$ can be considered as constant in a neighborhood of $\mathbf{s}$;

- there is a function of three arguments $C^S(\mathbf{h};\mathbf{s}_i,\mathbf{s}_i)$ such that $C(\mathbf{s}_i,\mathbf{s}_i) = C^S(\mathbf{s}_i - \mathbf{s}_i;\mathbf{s}_i,\mathbf{s}_i)$ and such that, for a given $\mathbf{h}$, $C^S(\mathbf{h};\mathbf{s}_i,\mathbf{s}_i)$ is a very regular and slowly varying (in the same sense as for $m_{\mathbf{s}}$) function of the two arguments $\mathbf{s}_i$ and $\mathbf{s}_j$. In other words, for locations $\mathbf{s}_i$ and $\mathbf{s}_j$ not too far from each other, $C^S(\mathbf{h};\mathbf{s}_i,\mathbf{s}_i)$ only depends on $\mathbf{h}$, and it is as if the covariance function $C(\mathbf{s}_i,\mathbf{s}_i)$ were locally stationary.

The key idea about *local stationarity* is that, even though process $\mathbf{Z_s}$ is globally non-stationary in $D \subset \mathbb{R}^d$, at any location $\mathbf{s} \in D$ there exists a neighbourhood $v_{\mathbf{s}}$ s.t.

$\mathbf{Z_s}$ restricted to $\nu_\mathbf{s}$ can be considered as stationary.
We suppose we can define $\nu_{\mathbf{s_0}} = \{s \in D : \|\mathbf{s_0} - \mathbf{s}\| \leq b\}$ wherein the mean function $m_\mathbf{s}$ and the covariance function $C(\mathbf{s}_i, \mathbf{s}_i)$ are approximately stationary, and wherein the available information is sufficient to make inference.

$$m_{\mathbf{s_0}} \approx m_{\mathbf{s_i}} \approx m_{\mathbf{s_j}}, \quad \forall \mathbf{s_0} \in D, \ \forall (\mathbf{s}_i, \mathbf{s}_i) \in \nu_{\mathbf{s_0}} \times \nu_{\mathbf{s_0}}$$

The terms "slowly varying" and "approximately constant" are not well defined, we are going to have some paramters that govern this variability, the coice of these parameters strongly influences the estimates of the covariance structure of a random process.

2.2 Anisotropy

Any isotropic covariance function can be turned to anisotropic by introducing the isotropy matrix $\mathbf{\Sigma}$ that takes into account geometric anisotropy:

$$C(\mathbf{s}_i, \mathbf{s}_j) = C\left(\sqrt{(\mathbf{s}_i - \mathbf{s}_j)^T \mathbf{\Sigma}^{-1} (\mathbf{s}_i - \mathbf{s}_j)}\right)$$

where with C we indicate the corresponding isotropic covariogram. In general we do not cover any isotropic process by "stretching and rotating" the plane with a symmetric positive definite matrix, but since $\mathbf{\Sigma_s}$ is going to be a function of space it is often enough in this context. We have the following covariance properties at any location $\mathbf{s_0} \in D : \forall (\mathbf{s}_i, \mathbf{s}_i) \in \nu_{\mathbf{s_0}} \times \nu_{\mathbf{s_0}}$:

$$C(\mathbf{s}_i, \mathbf{s}_j) \approx \sigma^2_{\mathbf{s_0}} R^S\left(\sqrt{(\mathbf{s}_i - \mathbf{s}_j)^T \mathbf{\Sigma}^{-1}_{\mathbf{s_0}} (\mathbf{s}_i - \mathbf{s}_j)}\right) \equiv C^S(\mathbf{s}_i - \mathbf{s}_j, \mathbf{s_0})$$

Where $R^S(\cdot)$ is an isotropic correlation function. Thus at any given position of a neighbohood moved around in the domain of interest, the mean $m_\mathbf{s}$ is constant and the non-stationary covariance function $C(\mathbf{s}_i, \mathbf{s}_j)$ is reduced to the anisotropic covariance function given in the equation above.
The anisotropy matrix can be parametrized according to spectral decomposition $\mathbf{\Sigma_s} = \mathbf{\Phi_s \Lambda_s \Phi_s^T}$, where Λ_s is the diagonal matrix of eigenvalues, and Φ_s is the matrix of eigenvectors:

$$\mathbf{\Lambda_s} = \begin{pmatrix} \lambda^2_{1\mathbf{s}} & 0 \\ 0 & \lambda^2_{2\mathbf{s}} \end{pmatrix}, \quad \mathbf{\Phi_s} = \begin{pmatrix} cos\phi_s & sin\phi_s \\ -sin\phi_s & cos\phi_s \end{pmatrix}, \quad \lambda_{1s}, \lambda_{2s} > 0, \, \phi_s \in [0, \frac{\pi}{2})$$

We are left with the following parameter functions to be estimated in any location $\mathbf{s} \in D$: $\sigma_\mathbf{s}$, $\lambda_{1\mathbf{s}}$, $\lambda_{2\mathbf{s}}$, $\phi_\mathbf{s}$, $m_\mathbf{s}$.

2.3 Variogram kernel estimator

Now we present what is the most important idea in (1) and we are going to understand why *local stationarity* is so useful in practice. The problem of the usual

variogram estimator is that, since 2 points are not guarranteed to have the same mean, the estimator becomes biased. But if we restrict ourselves to pairs of points $(\mathbf{s}_i, \mathbf{s}_j) \in \nu_{\mathbf{s_j}} \times \nu_{\mathbf{s_i}}$, we can say $\mathbf{Z_{s_i}}$ and $\mathbf{Z_{s_j}}$ will have approximately the same mean $m_{s_i} \approx m_{s_j}$ and we can resort to a slight variation of the correponding stationary variogram estimator that also takes into account the variation of the covariance function in each point of the domain and geometrical anisotropy.
Since we expect that the covariogram function, fixed $\mathbf{s_0} \in D$ as a "center of observation", if $C^S(\mathbf{s}_i - \mathbf{s}_j, \mathbf{s_0})$ is a corresponding stationary covariogram $\forall (\mathbf{s}_i, \mathbf{s}_j) \in \nu_{\mathbf{s_0}}$, that slowly varies in $\mathbf{s_0}$, we need to estimate a different empiric variogram in each point of our domain D. This method only considers pairs of points that are close to each other, if two points are too far their squared difference will not count in the sample mean, but if we center ourselves in $\mathbf{s}_0$ and there is a pair of locations $(\mathbf{s}_i, \mathbf{s}_j) : \|\mathbf{s}_i - \mathbf{s}_j\| \leq b$ we will not completely throw that information away, but we will weigh it through some kernel function.
Under this local stationarity framework, a non-parametric kernel moment estimator of the local stationary variogram $\gamma(\mathbf{h}, \mathbf{s}_0) = \sigma^2(\mathbf{s}_0) - C^S(\mathbf{h}; \mathbf{s}_0)$ in any fixed location $\mathbf{s}_0 \in D$ and for each spatial tolerance region T_k surrounding $\mathbf{h}$, $\|h\| \leq b$ is defined as follows:

$$\hat{\gamma}(T_k, \mathbf{s}_0) = \frac{\sum_{V_k} K_\epsilon^*(\mathbf{s}_0, \mathbf{s}_i) K_\epsilon^*(\mathbf{s}_0, \mathbf{s}_j) [Z_{\mathbf{s_i}} - Z_{\mathbf{s_i}}]^2}{2 \sum_{V_k} K_\epsilon^*(\mathbf{s}_0, \mathbf{s}_i) K_\epsilon^*(\mathbf{s}_0, \mathbf{s}_j)}$$

with:

$$V_k = \{(\mathbf{s}_i, \mathbf{s}_j) \in D \times D : \mathbf{s}_i - \mathbf{s}_j \in T_k\}$$

$$K_\epsilon^*(\mathbf{s}_0, \mathbf{s}_i) = \frac{K_\epsilon(\mathbf{s}_0, \mathbf{s}_i)}{\sum_{k=1}^n K_\epsilon(\mathbf{s}_0, \mathbf{s}_k)}$$

How T_k is built in practice is discuessed in the sections 2.4 and 3.4, but since we are considering the anisotropic case we can no longer just use intervals but we will have vector neighbourhoods.
Note that $\mathbf{s}_0$ does not need to belong to the locations where we know a realization of $Z_\mathbf{s}$, it can be any point in D; how we choose the locations where we estimate is discussed in section 2.6.
$K_\epsilon(\cdot, \cdot)$ is a non-negative, symmetric kernel on $\mathbb{R}^d \times \mathbb{R}^d$, with bandwidth parameter ϵ. The kernel function is used to smoothly weigh the squared differences (for each neighbourhood of spatial lags) according to the distance of these paired values from a target location. We assign to each data pair a weight proportional to the product of the individual weights. Observation pairs close to the target location have more influence on the locally stationary variogram estimator than those which are distant.
If $K_\epsilon(\cdot, \cdot) = 1$ and b is chosen as cutoff threshold we get the usual global stationary second-order variogram. In theory any kernel function could work for this task, but the authors suggest the use of an isotropic stationary Gaussian kernel: $K_\epsilon(\mathbf{s}_i, \mathbf{s}_j) = exp(-\frac{1}{2\epsilon^2}\|\mathbf{s}_i - \mathbf{s}_j\|^2)$, since it has a non-compact support and therefore includes all observations. This avoids artifacts caused by the sole use of observations close to the target locations. It also reduces the instability of the local stationary variogram kernel estimator computed at regions with low sampling density and provides a smooth parameter estimate to make it compatible with the local stationarity assumption. In

our code structure it is possibile to easily add new kernel functions and choose them through the R interface. The parameter b, the radius of local stationarity neighbourhoods, is chosen as a function of the bandwidth parameter ϵ, $b = 2\epsilon$ in our code, but in can obviously be easily changed. The code that calculates the empiric variogram is presented in section 3.5.

2.4 The grid

The sub-intervals for empiric variogram estimation now become vector regions to take into account anisotropy. Each pair $(\mathbf{s}_i, \mathbf{s}_j) \in D \times D \; s.t. \; \|\mathbf{s}_i - \mathbf{s}_j\| \leq b$ needs to be assigned to some tolerance region T_k.

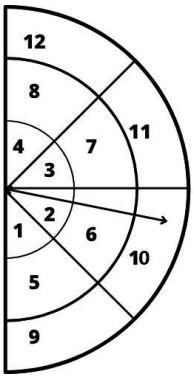


Figure 2.1: Visualization of the construction of vector neighbourhoods

We fix the number of directions of anisotropy *n_angles* (4 in the example image) and the number of sub-intervals *n_intervals* (3 in the example image) and we get a total number of $H = n\_angles \times n\_intervals$ of vector neighbourhoods. Each vector $\mathbf{s}_i - \mathbf{s}_j \in D \times D \; s.t. \; \|\mathbf{s}_i - \mathbf{s}_j\| \leq b$ is centered in the grid and is assigned an index just as shown in the picture (It would be 10 in the example), the full algorithm is presented in 3.4. A semicircle is sufficient because any pair can be made to fit into it, if it doesn't already, by taking the opposite.

2.5 WLS fitting of parameters

The proposed fitting criterion for the empiric variogram is weighted leas squares (WLS) at each position $\mathbf{s}_0 \in D$ where we calculated the empiric variogram. We are going to present this method in a formally different but practically equivalent way from what is described in (1).
At any location $\mathbf{s}_0 \in D$, the estimation of the parameters vector $\boldsymbol{\theta}_{\mathbf{s}_0} = \{\sigma_{\mathbf{s}_0}, \lambda_{1\mathbf{s}_0}, \lambda_{2\mathbf{s}_0}, \phi_{\mathbf{s}_0}\}$ characterizing the local stationary spatial dependece structure $\gamma(\cdot, \mathbf{s}_0) \equiv \gamma(\cdot, \boldsymbol{\theta}_{\mathbf{s}_0})$

is given by the following minimization problem:

$$\hat{\boldsymbol{\theta}}_{\mathbf{s_0}} = \underset{\boldsymbol{\theta}\in\Theta}{\arg\min} \sum_{k=1}^{H} w_k(\mathbf{s}_0)(\gamma(T_k;\boldsymbol{\theta}) - \hat{\gamma}(T_k;\boldsymbol{\theta}))^2$$

H is the total number of vector neighbourhoods as defined in previous section. Θ is an open parameter space and $\boldsymbol{\theta}$ is the vector of unkwnow parameters. T_k is the k-th vector region indexed as in Figure 1, and by $\gamma(T_k;\boldsymbol{\theta})$ we mean a valid variogram model γ with parameters $\boldsymbol{\theta}$ evaluated in a location given by averaging all spatial data pairs whose difference belongs to tolerance region T_k. $\hat{\gamma}(T_k;\boldsymbol{\theta})$ Is the k-th component of the binned variogram we already calulated in section 2.3.

$$w_k(\mathbf{s}_0) = \frac{\sum_{V_k} K_c^*(\mathbf{s}_0,\mathbf{s}_i)K_c^*(\mathbf{s}_0,\mathbf{s}_j)}{\|T_k\|}$$

are the weights of our WLS problem, the numerator is what we had as denominator in the variogram kernel estimator, and by $\|T_k\|$ we mean the average of distances of all pairs of spatial locations belonging to V_k.
The numerical optimization method used for fitting the variogram we used is L-BFGS-B, a limited memory variation of the BFGS quasi-newton method with box constraints, proposed in (9), well suited for our nonlinear weighted least squares problem.

2.6 Anchor points and Smoothing

Up untill now we have shown how to estimate the empiric variogram and fit a variogram model in a fixed position $\mathbf{s}_0 \in D$. In order to get the full covariance structure in D we just need to perform the same procedure in a fixed discrete subset of locations in D and then perform a kernel smoothing of the parameter vector $\boldsymbol{\theta}_{\boldsymbol{s}}$. The points where we estimate and fit the variogram are called *anchor points* and we can choose them in many different ways, in implementation we let the user decide an upper bound for the numerosity of anchor points n^2 which defines the computational burden of the execution of the program, and then we take equally spaced locations in out domain as described in section 3.2 (as suggested by the authors in (1)).
The Nadaraya-Watson kernel estimator of the parameter vector $\boldsymbol{\theta}_{\mathbf{s}} = \{\sigma_{\mathbf{s}}, \lambda_{1\mathbf{s}}, \lambda_{2\mathbf{s}}, \phi_{\mathbf{s}}\}$ at any location $\mathbf{s} \in D$ is given by:

$$\tilde{\boldsymbol{\theta}}_{\mathbf{s}} = \sum_{k=1}^{m} W_k(\mathbf{s})\boldsymbol{\theta}(\mathbf{s_k}),\ W_k(\mathbf{s}) = \frac{K(\frac{\mathbf{s}-\mathbf{s_k}}{\delta})}{\sum_{k=1}^{m} K(\frac{\mathbf{s}-\mathbf{s_k}}{\delta})}$$

where m is the total number of anchor points indexed by k. K is a d-variate kernel function, in our implementasion a Gaussian isotropic stationary kernel, as for the variogram estimate, because parameters functions are assumed to vary slowly and regularly across the domain D. The choice of the smoothing bandwidth δ associated

with the parameter function $\sigma_{\mathbf{s}}$ is carried out using the following cross-validation criterion (3):

$$\delta = \arg\min_{\delta \in \Delta} \frac{1}{m} \sum_{k=1}^{m} \left(\frac{\hat{\sigma}(\mathbf{s_k}) - \tilde{\sigma}(\mathbf{s_k})}{1 - W_k(\mathbf{s_k})} \right)$$

Δ is an interval that can either be defined by the user or as a function of ϵ.
Theoretically a different bandwith parameter δ could be chosen for each parameter in θ, but numerical simultaions show that choosing it for σ and using it for all other parameters makes very little difference in terms of performance.

2.7 Prediction - Kriging

Now that we have the whole covariance structure and paramteres in D the remaining parameter left to be estimated is $m_{\mathbf{s}}$. Since it can be considered constant within the quasi-stationary neighborhood, m_s can be estimated explicitly by a kriging of the mean based on the local stationary spatial dependence structure whose parameters have already been estimated:

$$\hat{m_{\mathbf{s}}} = \sum_{\mathbf{s_i} \in \nu_{\mathbf{s}}} \eta_i(\mathbf{s}) Z_{\mathbf{s_i}}$$

where $\boldsymbol{\eta}(\mathbf{s}) = [\eta_i(\mathbf{s})]$ are the kriging weights for the mean given by

$$\boldsymbol{\eta}(\mathbf{s}) = \frac{\hat{\mathbf{\Gamma}}_{\mathbf{s}}^{-1}\mathbf{1}}{\mathbf{1}^T \hat{\mathbf{\Gamma}}_{\mathbf{s}}^{-1}\mathbf{1}}$$

with $\hat{\mathbf{\Gamma}}_{\mathbf{s}} = [\gamma(\mathbf{s_i} - \mathbf{s_j}; \hat{\boldsymbol{\theta}}(\mathbf{s}))]$, $(\mathbf{s_i}, \mathbf{s_j}) \in \nu_{\mathbf{s}} \times \nu_{\mathbf{s}}$. Given the vector of a realization of a random process $\boldsymbol{Z} = (Z(\mathbf{s_1}), ..., (Z(\mathbf{s_n}))^T$, the point predictor for the unknown value of the random process $Z_{\mathbf{s}}$ at an unsampled location $\mathbf{s} \in D$ is given by the optimal linear predictor:

$$\hat{Z}_{\mathbf{s}} = m_s + \sum_{i=1}^{n} \eta_i(\mathbf{s})(Z_{\mathbf{s_i}} - m_{\mathbf{s}_i})$$

where the prediction kriging vector $\boldsymbol{\eta}(\mathbf{s}) = [\eta_i(\mathbf{s})]$ and the corresponding kriging variance $Q_{\mathbf{s}}$ are given by $\boldsymbol{\eta}(\mathbf{s}) = \mathbf{C^{-1}C_0}$ and $Q_{\mathbf{s}} = \sigma_s - \mathbf{C_0^T C^{-1} C_0}$, with $\mathbf{C}_0 = [C(\mathbf{s}_i - \mathbf{s}); \hat{\boldsymbol{\theta}}(\mathbf{s}))]$; and $\mathbf{C} = [C(\mathbf{s}_i - \mathbf{s}_j); \hat{\boldsymbol{\theta}}(\mathbf{s}))]$. The bandwidth parameter ϵ can either be chosen arbitrarily by the user or, now that we have a prediction method, chosen by cross-validation w.r.t. MSE.

Chapter 3

Structure of the code

In this chapter we will have a detailed view on how we have implemented the algorithm of chapter 2 in C++, learn how every class works and which are its purposes.

3.1 Introduction

First, let's have a brief look at *traits.hpp* and learn how data is stored. Every location is identified by the vector of its coordinates in the plane. Since theoretically the package could be expanded to more than two dimensions, we will use an Eigen dynamic vector to store them. All the matrices computed by the algorithm are Eigen dynamic matrices of doubles or integers stored dynamically via shared pointers, since they can be huge, and we need to grant to different objects access to them without creating duplicates.

```
// defining basic types
using vector = Eigen::VectorXd;
using matrix = Eigen::MatrixXd;
using matrixI = Eigen::MatrixXi;
using vectorptr = std::shared_ptr<vector>;
using matrixptr = std::shared_ptr<matrix>;
using matrixIptr = std::shared_ptr<matrixI>;
using vectorind = std::vector<size_t>;
```

Eventually we have defined two different types of standard functions: one to represent the functions used in the kernel and one to represent the functions used to build the grid. A better explanation for them is provided later.

```
// defining function types
using kernelfunction = std::function<double(const vector&, const
    vector&, const double&)>;
using gridfunction = std::function<matrixIptr(const matrixptr&,
    const size_t&, const size_t&, const double&)>;
```

3.2 Anchor points

The **Anchor class** works independently from the rest of the code and its only purpose is, given a set d of coordinates, find the position of anchor points equally spaced on the regions of the plane identified by the locations in d. The function `find_indeces` draws a grid over the points in d and associates each one with the number of his cell.

```
/**
 * \brief return the index of the position in the grid of each of
     the points of the dataset "m_data"
 */
Eigen::VectorXi find_indeces()
{
    ...
}
```

Algorithm 1 find_indeces

Let n be the number of rows of d
Let m be the number of desired tiles per row and column
$c_x = min(x_i) \forall i \leq n$
$c_y = min(y_i) \forall i \leq n$
$W = max(x_i) - c_x$
$H = max(y_i) - c_y$
$w = \frac{W}{m}$
$h = \frac{H}{m}$
for every $i \leq n$ **do**
$p_i = \lceil \frac{x_i - c_x}{w} \rceil + m * \lfloor \frac{y_i - c_y}{h} \rfloor$ where p_i is the position on the grid of the i-th point
end for

Eventually the function `find_anchorpoints` returns the coordinates of the center of each cell such that at least one of the location in d belongs to it.

```
/**
 * \brief this function returns the coordinates of the anchor points
     in a way such that every anchor point has at
 * least one point of the domain in its neighbourhood
 */
const cd::matrix find_anchorpoints()
{
    ...
}
```

Algorithm 2 find_anchorpoints

Let r be the vector of integers returned by the function `find_indeces`
Let p be a new empty vector of integers
for every component i of the vector r **do**

if i does not already belong to p **then**
Add i to p
end if
end for
for every integers i inside the vector p **do**
create a new anchor point (x_i, y_i) such that $x_i = c_x + (i - \lfloor \frac{i}{m} \rfloor * m) * w - \frac{w}{2}$ and $y_i = c_y + \lceil \frac{i}{n\_tiles} \rceil * h - \frac{h}{2}$
end for

3.3 The kernel

The **Kernel class** answers two different purposes:

- **Computing** the kernel function between two points of the plane.
- **Storing** somewhere the value of the function at point (1) evaluated between the locations whose coordinates are inside a specific dataset. This is particularly useful when we compute the sample variogram and when we perform smoothing on the parameters of the fitted one.

In order to accomplish the first goal, we have decided to write a **factory function** which returns the function corresponding to the string taken as input and is called by the constructor of the class which allocates in `f` the one selected. This way the user can pass a string to the constructor and choose the function that better suits his model. This functions can be found inside *kernelfunctions.hpp*.

```
/**
 * \return e^(-norm(x-y)^2/epsilon^2)
 */
double gaussian(const cd::vector& x, const cd::vector& y, const
    double& epsilon);

/**
 * \return 1 only if the norm of the difference between x and y is
     less the the square of epsilon
 */
double identity(const cd::vector& x, const cd::vector& y, const
    double& epsilon);

/**
 * \brief allow to select between pre-built kernel functions
 * \param id a string with the name of the kernel function you want
     to use
 */
cd::kernelfunction make_kernel(const std::string& id);
```

Since we need to compute `f` between the same points multiple times, we have added as a member `m_k`, a shared pointer to an Eigen dynamic matrix which can

be filled in two different ways: calling the function `build_kernel` or one of the two versions of the function `build_simple_kernel`.

```
/**
 * \brief build the "star" version of the kernel that contains the
     standardized kernel weights in such
 * a way that each row sums to one
 * \param data a shared pointer to the matrix with the coordinates
     of the original dataset
 * \param anchorpoints a shared pointer to the matrix with the
     coordinates of the anchor points
 */
void build_kernel(const cd::matrixptr& data, const cd::matrixptr&
    anchorpoints);

/**
 * \brief build the "standard" version of the kernel needed for
     smoothing
 * \param coordinates a shared pointer to the matrix with the
     coordinates
 */
void build_simple_kernel(const cd::matrixptr& coordinates);

/**
 * \brief build the "standard" version of the kernel needed for
     smoothing
 * \param coordinates a shared pointer to the matrix with the
     coordinates
 * \param epsilon replace the old epsilon with a new value
 */
void build_simple_kernel(const cd::matrixptr& coordinates, const
    double& epsilon);
```

The first one fills $k(i, j)$ with `f` evaluated between the ith anchor point and the jth point of `coordinates`, then divides each component of k with the sum of the coefficients on the same row. The second ones simply fill $k(i, j)$ with $f(i, j)$ where i and j are the ith and jth points of `coordinates` and possibly change the value of ε.

3.4 The grid

The **Grid class** solves the specific goal of computing and storing the grid of positions of the vectors linking each pair of points in a dataset. In addition, it computes and stores dynamically via shared pointers the coordinates of the centre and the norm of each cell of the grid. In our implementation, the x, y and norm of the tile n are obtained by sample mean of the x, y and norm of all the vectors which fell in the nth cell via the function `build_normh`.

```
/**
```

```
 * \brief a "helper" function to build the vector containing the
     position of the centers of the cells of the grid.
 * Each pair of coordinates is assigned to a position of the grid.
 * \param data a shared pointer to the matrix of the coordinates
 */
void build_normh(const cd::matrixptr& data);
```

Algorithm 3 build_normh

Set H equal to maximum coefficient in g.
Initialize X, Y and normh to 0^H, where 0^H is a zero vector of lenght H.
Initialize nn to 0^H.
for every couple of points i and j **do**
  Set k equal the position of the vector i - j in the grid
  $normh_k = normh_k + \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
  if i - j belongs to the first or the third quadrant **then**
    $X_k = X_k + |x_i - x_j|$
  else
    $X_k = X_k - |x_i - x_j|$
  end if
  $Y_k = Y_k + |y_i - y_j|$
  $nn_k = nn_k + 1$
end for
for every component u of X, Y and normh **do**
  if $nn_u \neq 0$ **then**
    $X_u = \frac{X_u}{nn_u}$
    $Y_u = \frac{Y_u}{nn_u}$
    $normh_u = \frac{normh_u}{nn_u}$
  end if
end for

The package contains our personal implementation of a **grid generator** which builds a pizza-like structure taking as input a certain number of angles and a certain number of intervals per angle.

```
/**
 * \brief this function builds a 2D-grid using a "a fette di pizza"
     (slices-of-pizza like) algorithm to partition
 * the domain
 * \param data a shared pointer to the matrix of the coordinates
 * \param n_angles number of slices of the pizza
 * \param n_intervals number of the pieces for each slice of the
     pizza
 * \param epsilon bandwidth parameter epsilon. Same of the kernel
 */
cd::matrixIptr pizza(
    const cd::matrixptr& data, const size_t& n_angles, const size_t&
        n_intervals, const double& epsilon);
```

Algorithm 4 Pizza

Suppose n is the number of row of the matrix d.
Create a new $n \times n$ matrix G and fill it with -1.
Set $b = 2 * \epsilon$
Set $cell\_length = \frac{b}{n\_intervals}$
Set $cell\_angle = \frac{\pi}{n\_angles}$
for every couples of points i and j **do**
  $\Delta x = x_j - x_i$
  $\Delta y = y_j - y_i$
  $r = \sqrt{\Delta x^2 + \Delta y^2}$
  if $r > b$ **then**
    do nothing
  else if $\Delta x = 0$ **then**
    $G(i,j) = \lfloor \frac{r}{cell\_length} \rfloor$
  else
    $G(i,j) = \lfloor \frac{r}{cell\_length} \rfloor + n\_intervals * \lfloor \frac{\pi}{2} + \frac{\arctan \frac{\Delta x}{\Delta y}}{cell\_angle} \rfloor$
  end if
end for

Since a different algorithm may be preferred in alternative to ours, we have implemented inside *gridfunctions.hpp* a structure similar to the one in *kernelfunctions.hpp* with a **factory function** which can be expanded to accept new user-defined functions.

```
/**
 * \brief allow to select between the preferred method to build the
     grid
 * \param id name of the function of choice
 */
cd::gridfunction make_grid(const std::string& id);
```

3.5 The sample variogram

The **SampleVar class** is the backbone of the entire algorithm. It builds and stores a Grid object and a Kernel object which are then used to calculate the sample variogram in each anchor point. This is all done by the function `build_samplevar` which does exactly what the name suggests.

```
/**
 * \brief build the matrix of the empiric variogram
 * \param data a shared pointer to the matrix of the coordinates of
     the original dataset
 * \param anchorpoints a shared pointer to the matrix of the
     coordinates of the anchor poitns
 * \param z a shared pointer to the vector of the value of Z
```

```
 */
void build_samplevar(const cd::matrixptr& data, const cd::matrixptr&
    anchorpoints, const cd::vectorptr& z);
```

Algorithm 5 build_samplevar

Let K be the kernel matrix and g the grid matrix.
Let H be the maximum coefficient of g, N the number of anchor points and n the number of points of d.
Create two new $H \times N$ matrices V and D.
Create a new matrix Z
for every couple of points in d **do**
  $Z(i,j) = (z_i - z_j)^2$;
  $Z(j,i) = Z(i,j)$;
end for
for every anchor point l **do**
  Initialize counter to 0^H
  for every couple of points i and j in d **do**
    Set k equal to the position of the vector i - j in the grid
    $V(k,l) = V(k,l) + K(l,i) * K(i,l) * Z(i,j)$
    $D(k,l) = D(k,l) + K(l,i) * K(i,l)$
    $counter_k = counter_k + 1$
  end for
  for every component u of the vector counter **do**
    if $counter_u \neq 0$ **then**
      $V(u,l) = \frac{V(u,l)}{2*D(u,l)}$
    end if
  end for
end for

In addition, it calls the private function `build_squaredweights` which has the purpose of calculating and storing inside the matrix pointed by `squaredweights` the weights required to solve the non-linear optimization problem in the next step.

```
/**
 * \brief a "helper" function which built the squared weights for
     the wls problem needed by the optimizer
 */
void build_squaredweights();
```

Algorithm 6 build_squareweights

Let g be the kernel matrix, $normh$ the vector with the norm of each cell, H the length of normh and N the number of columns of the matrix D built via `build_samplevar`

Create a new $N \times H$ matrix W

for every $k \leq N$ and every $h \leq H$ **do**
$W(k,h) = \frac{D(h,k)}{normh_h}$
end for

3.6 Fit the variogram

The **class Opt** has been written with the sole purpose of solving the **non-linear optimization problem** of fitting the sample variogram with a specific variogram function decided by the user which depends on a vector of parameters. This is solved for each anchor point and the solutions are stored in a matrix. To accomplish this goal, we rely on the library LBFGSpp which implements the limited-memory version of the Broyden–Fletcher–Goldfarb–Shanno algorithm with box constraints. This library has been chosen for different reasons:

- The method BFGS in the limited-memory version is particularly suited to problems with very large numbers of variables and it is **very fast** since its computational complexity is only $O(n^2)$, compared to $O(n^3)$ in Newton's method.
- It is a header-only library, and it can be **shipped directly with the package.**
- It is very **simple** to use.

In order to interface with the optimization library, we have written the functor **TargetFunction** which presents two different versions of the overload of the `operator()`.

```
/**
 * \param params a vector containing the previous value of the
     parameters of the function (lambda1, lambda2, phi, sigma, etc.)
 * \param grad a vector containing the previous value of the
     gradient which is updated at each iteration
 */
double operator() (const cd::vector& params, cd::vector& grad);

/**
 * \param params a vector containing the previous value of the
     parameters of the function (lambda1, lambda2, phi, sigma, etc.)
 */
double operator() (const cd::vector& params);
```

The second one simply computes the value of the function f given a vector of parameters and a couple of coordinates. The first one instead, does the same thing, but at the same time updates the value of the gradient at each iteration during the optimization.

3.7 The variogram functions

To allow the choice between different variogram functions we have implemented a **factory** which exploits **polymorphism** returning a pointer to an object of **class VariogramFunction**, from which all different types of variograms are derived.

```
class VariogramFunction {
protected:
    /**
     * \brief convert the isotropic variogram in the equivalent
         anisotropic one calculating the norm of the spatial lag
     * rotated and expanded according to the eigenvalues and
         eigenvector of the anisotropy matrix
     */
    double compute_anisotropic_h(
        const double& lambda1, const double& lambda2, const double&
            phi, const double& x, const double& y);

public:
    VariogramFunction() = default;
    /**
     * \brief return f(params, x, y)
     */
    virtual double operator()(const cd::vector& params, const
        double& x, const double& y) = 0;
}; // class VariogramFunction

class Exponential : public VariogramFunction {
public:
    Exponential() = default;
    /**
     * \return sigma * sigma * (1 - exp(-h))
     * \param params a vector with lambda1, lambda2, phi and sigma
         in this exact order
     */
    double operator()(const cd::vector& params, const double& x,
        const double& y) override;
}; // class Exponential

class Matern : public VariogramFunction {
public:
    Matern() = default;
    /**
     * \return sigma * sigma *(1 - std::pow(std::sqrt(2*nu)*h,
         nu)*std::cyl_bessel_k(nu,
     * std::sqrt(2*nu)*h)/(std::tgamma(nu)*std::pow(2,nu-1)))
     * \param params a vector with lambda1, lambda2, phi, sigma and
         nu in this exact order
     */
    double operator()(const cd::vector& params, const double& x,
        const double& y) override;
}; // class Matern

class MaternNuFixed : public VariogramFunction {
private:
    double m_nu = 0.5; ///< constant value of nu
```

```
public:
    MaternNuFixed(const double& nu)
        : m_nu(nu) {};
    /**
     * \return sigma * sigma *(1 - std::pow(std::sqrt(2*nu)*h,
         nu)*std::cyl_bessel_k(nu,
     * std::sqrt(2*nu)*h)/(std::tgamma(nu)*std::pow(2,nu-1)))
     * \param params a vector with lambda1, lambda2, phi and sigma
         in this exact order
     */
    double operator()(const cd::vector& params, const double& x,
        const double& y) override;
}; // class MaternNuFixed

class Gaussian : public VariogramFunction {
public:
    Gaussian() = default;
    /**
     * \return sigma * sigma * (1 - exp(-h*h))
     * \param params a vector with lambda1, lambda2, phi and sigma
         in this exact order
     */
    double operator()(const cd::vector& params, const double& x,
        const double& y) override;
}; // class Gaussian

/**
 * \brief allow to select between different functions for the
     variogram
 * \param id the name of chosen variogram
 */
std::shared_ptr<VariogramFunction> make_variogramiso(const
    std::string& id);
```

The class VariogramFunction implements the method `compute_anisotropic_h` which compute the value of the parameter h that converts the isotropic variogram in the equivalent anisotropic one calculating the norm of the spatial lag rotated and expanded according to the eigenvalues and eigenvector of the anisotropy matrix. All the classes provide an overload of the `operator()` and via the factory function are then wrapped inside TargetFunction.

3.8 Smoothing

Once solved the optimization problem for the anchor points, the **Smt class** allows you to compute the value of the parameters, and so of the variogram function, in any point of the plane you want to. In order to do this the class relies on a member of the class Kernel. As we have seen before the kernel required a specific value of ε to build the kernel matrix and to evaluate the kernel function between two points.

In the case of the smoothing this value is called δ and its optimal value is calculated by the **constructor** of the class via **cross-validation**. A second constructor has been implemented to perform smoothing when we already know the best value for δ.

```
/**
 * \brief constructor
 * \param solutions a shared pointer to the solutions of the
     optimization
 * \param anchorpos a vector containing the indeces of the anchor
     position obtained by clustering
 * \param d a shared pointer to the matrix of the coordinates
 * \param min_delta the minimum exponent for the cross-validation of
     the delta bandwidth parameter for gaussian
 * kernel smoothing \param max_delta the maximum exponent for the
     cross-validation of the delta bandwidth parameter
 * for gaussian kernel smoothing
 */
Smt(const cd::matrixptr& solutions, const cd::matrixptr& anchorpos,
    const double& min_delta,
   const double& max_delta, const std::string& kernel_id);
/**
 * \brief constructor
 * \param solutions a shared pointer to the solutions of the
     optimization
 * \param anchorpos a vector containing the indeces of the anchor
     position obtained by clustering
 * \param d a shared pointer to the matrix of the coordinates
 * \param delta a user-chosen value for delta
 */
Smt(const cd::matrixptr& solutions, const cd::matrixptr& anchorpos,
    const double delta,
   const std::string& kernel_id);
```

Algorithm 7 constructor

Set $min\_error = +\infty$
for Every δ between min_delta and max_delta **do**
  Let K be the kernel matrix built using delta
  Set $error = 0$
  for every anchor point j **do**
    Let σ_p be the value of σ predicted in j
    Let σ_r be the real value of sigma in j
    $error = error + \frac{(\sigma_p - \sigma_r)^2}{(\frac{1-K(j,j)}{\sum_i K(j,i)})^2}$
  end for
  if $error \leq min\_error$ **then**
    $optimal\_delta = \delta$
    $min\_error = error$
  end if

end for

This class relies on **templates** in order to perform smoothing both on new points provided by the user and on the anchor points used to build the model when we search for the optimal δ.

```
/**
 * \brief smooth all the parameters for a point in position pos
 * \param pos a vector of coordinates or the index of the position
     of the point where to find the smoothed value of the parameters
 */
template<class Input>
cd::vector smooth_vector(const Input& pos) const
{
    cd::vector result(m_solutions->cols());
    for (unsigned int i=0; i<m_solutions->cols(); ++i)
        result(i) = smooth_value(pos, i);
    return result;
};
```

3.9 Kriging

Finally, the algorithm performs **kriging** on the mean and the punctual value of the function f via an object of **class Predictor**. This class heavily relies on **templates** to perform kriging both on single points and on full databases.

```
/**
 * \brief predict the mean
 */
template<typename Input, typename Output>
Output predict_mean(const Input& pos) const;

/**
 * \brief predict Z
 */
template<typename Input, typename Output>
Output predict_z(const Input& pos) const;
```

3.10 Parallelization

We have payed special attention to the calculation **speed** of the package and we have used *OpenMP* to **parallelize** the thoughest sections of our code. The functions `build_kernel` and `build_simple_kernel` which build the kernel matrix, the function `Pizza` (4) that constructs the grid, the functions `build_samplevar` (5) and `build_squaredweights` (6) which calculate the sample variogram, the function `findallsolutions` inside the *opt class*, the constructor of the *smt class* and

eventually the functions `predict_mean` and `predict_y` for kriging, rely all on the parallelization in order to have a huge improvement in their performance.

⚠Beware that by default *OpenMP* will take advantage of all the cores of your processor, however is always possible to pass to each function the number of threads you want to use.

Here is a simple benchmark to better understand how *OpenMP* can speed up calculations. The table shows how two different processors performed with and without parallelization in the example of chapter 6 .For a better explanation on what these functions do and how you can use them, you can check the following chapters of the paper.

	AMD Ryzen™ 5 2600x		Intel® Core™ i7-10700KF	
	1 thread	12 threads	1 thread	16 threads
`variogram.lsm`	3 ms	1 ms	1 ms	1 ms
`findsolutions.lsm`	389 ms	189 ms	395 ms	122 ms
`plot.lsm`	4275 ms	1638 ms	3959 ms	413 ms
`predict.lsm`	1100 ms	151 ms	1020 ms	112 ms

Chapter 4

R/C++ Interface

In order to interface our precompiled C++ code with R, we heavily relied on the package RcppEigen, that provides an easy way to port R data structures to C++ and Eigen data types and structures.
The files are structured in the following way:

- **R**: The R folder contains all the R functions, mainly plot functions and also wrapper functions to define what the final user interface will be. It also contains the file "RcppExports.R" which is one of the three key files in the R/C++ interface (details below).
- **src**: It contains all the C++ files described in chapter 3 and also the files "RcppEigenInterface.cpp" and "RcppExports.cpp" described below.
- **man**: it contains the documentation of our R package, generated with roxygen.
- **data**: .Rdata files needed for the examples
- **DESCRIPTION**: brief description file of the package, it also contains contact details for issues about the package.
- **NAMESPACE**: file containing directives for the exports to the R environment.

Now we explain how the actual structure of the interface between R and C++ works, by presenting an example for a single function **variogramlsm**; it works exactly the same for every function present in the **RcppExports.R** file.

In **RcppExports.R** we have the definition of the **variogramlsm** function:

```
variogramlsm <- function(z, d, anchorpoints, epsilon, n_angles,
    n_intervals, kernel_id, print, n_threads)
    {.Call('_LocallyStationaryModels_variogramlsm', PACKAGE =
    'LocallyStationaryModels', z, d, anchorpoints, epsilon,
    n_angles, n_intervals, kernel_id, print, n_threads)}
```

The `.Call` function is an easy way to pass data types and data structures from R to C++ so the arguments of the R function and the objects passed to C++ are the same, but in order to have a different interface and different arguments of our R/C++ functions we create R wrapper functions, available for the user in the namespace. For example for **variogramlsm** the function in the namespace of the package **variogram.lsm**.
The `.Call` function passes all the arguments to the corresponding function in **src/RcppExports.cpp**:

```
// variogramlsm

RcppExport SEXP _LocallyStationaryModels_variogramlsm(SEXP zSEXP, SEXP
    dSEXP, SEXP anchorpointsSEXP, SEXP epsilonSEXP, SEXP n_anglesSEXP,
    SEXP n_intervalsSEXP, SEXP kernel_idSEXP, SEXP printSEXP, SEXP
    n_threadsSEXP) {
BEGIN_RCPP
    Rcpp::RObject rcpp_result_gen;
    Rcpp::RNGScope rcpp_rngScope_gen;
    Rcpp::traits::input_parameter< const Eigen::VectorXd& >::type
        z(zSEXP);
    Rcpp::traits::input_parameter< const Eigen::MatrixXd& >::type
        d(dSEXP);
    Rcpp::traits::input_parameter< const Eigen::MatrixXd& >::type
        anchorpoints(anchorpointsSEXP);
    Rcpp::traits::input_parameter< const double& >::type
        epsilon(epsilonSEXP);
    Rcpp::traits::input_parameter< const unsigned int& >::type
        n_angles(n_anglesSEXP);
    Rcpp::traits::input_parameter< const unsigned int& >::type
        n_intervals(n_intervalsSEXP);
    Rcpp::traits::input_parameter< const std::string& >::type
        kernel_id(kernel_idSEXP);
    Rcpp::traits::input_parameter< const bool >::type print(printSEXP);
    Rcpp::traits::input_parameter< const int& >::type
        n_threads(n_threadsSEXP);
    rcpp_result_gen = Rcpp::wrap(variogramlsm(z, d, anchorpoints,
        epsilon, n_angles, n_intervals, kernel_id, print, n_threads));
    return rcpp_result_gen;
END_RCPP
}
```

SEXP stands for pointers to SEXPREC or 'S expression' structures, the internal representation of R objects. The functions in **src/RcppExports.cpp** have the task to simply transform all the SEXP objects obtained through R to corresponding generic C++ or Eigen Types and then the function **variogramlsm** in **src/RcppEigenInterface.cpp** is called:

```
Rcpp::List variogramlsm(const Eigen::VectorXd &z, const Eigen::MatrixXd
    &d, const Eigen::MatrixXd &anchorpoints, const double& epsilon,
```

```
 const unsigned int& n_angles,
const unsigned int& n_intervals, const std::string &kernel_id, const
    bool print, const int &n_threads) {
// start the clock
auto start = high_resolution_clock::now();
// if n_threads is positive open open n_threads threads to process
    the data
// otherwise let openmp decide autonomously how many threads use
// if n_threads is greater than the maximum number of threads
    available open all the threads accessible
if (n_threads > 0)
{
    int max_threads = omp_get_max_threads();
    int used_threads = std::min(max_threads, n_threads);
    Rcpp::Rcout << "desired: " << n_threads << std::endl;
    Rcpp::Rcout << "max: " << max_threads << std::endl;
    Rcpp::Rcout << "used: " << used_threads << std::endl;
    omp_set_num_threads(used_threads);
}

matrixptr dd = std::make_shared<matrix>(d);
vectorptr zz = std::make_shared<vector>(z);
matrixptr anchorpointsptr = std::make_shared<matrix>(anchorpoints);

samplevar samplevar_(kernel_id, n_angles, n_intervals, epsilon);
// build the sample variogram
samplevar_.build_samplevar(dd, anchorpointsptr, zz);
// stop the clock and calculate the processing time
auto stop = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(stop - start);

if(print)
  Rcpp::Rcout << "task successfully completed in " <<
      duration.count() << "ms" << std::endl;

return
    Rcpp::List::create(Rcpp::Named("kernel")=*(samplevar_.get_kernel()),
                       Rcpp::Named("grid")=*(samplevar_.get_grid()),
                       Rcpp::Named("mean.x")=*(samplevar_.get_x()),
                       Rcpp::Named("mean.y")=*(samplevar_.get_y()),
                       Rcpp::Named("squaredweigths")=*(samplevar_.get_squaredweights()),
                       Rcpp::Named("empiricvariogram")=*(samplevar_.get_variogram()),
                       Rcpp::Named("anchorpoints")=anchorpoints,
                       Rcpp::Named("epsilon")=epsilon);}
```

Functions in **src/RcppEigenInterface.cpp** build all the desired objects using the rest of C++ code in src directory and return them as an Rcpp::List, objects in a list in R can be accessed through the usual $ operator.

Chapter 5

Examples

In this chapter we present 3 examples: the first one is an attempt to reproduce the example in (1), the other two are about the application of the same method to simulated non-stationary random processes.

5.1 Swiss rainfall data example

We tried to reproduce the results reported in (1) with the convolutional kernel method for quasistationary estimation implemented in our package. We faced some issues in getting the exact same results, because some choices of the authors were not reported. For example, even though the dataset is quite famous in geostatistical applications, a random validation split was performed and data actually used for covariance estimation was not indicated. Moreover, only anisotropy ratio was reported instead of the values of both eigenvalues, but by bounding their value to the bandwidth parameter, which means imposing $0 < \lambda_1 < \epsilon$ and $0 < \lambda_2 < \epsilon$, we obtained a result very similar to the one reported in the paper. For this example the cutoff parameter is set to $b = \sqrt{3}\epsilon$.

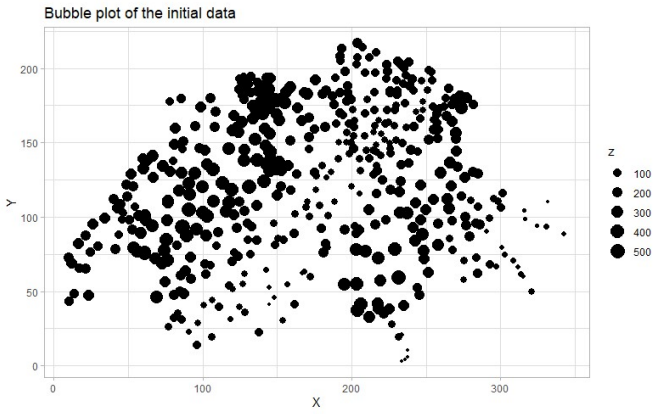


Figure 5.1: Swiss Rainfall Data 8th May 1986 in 467 fixed locations

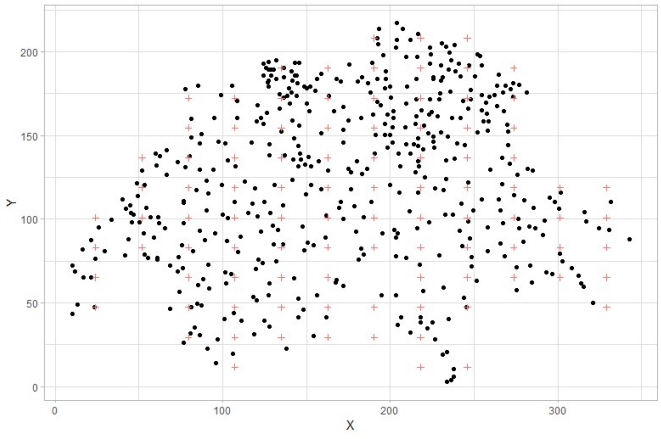


Figure 5.2: Anchor points in red

The bandwidth parameter epsilon is set to $\epsilon = 46$ km and the optimal smoothing

bandwidth associated to the Gaussian kernel corresponds to $\delta = 11.8$ km (in the paper the reported result is 11km).

```
    # Build the empiric variogram
vario <- variogram.lsm(z,d,a$anchorpoints,epsilon = 46,n_angles =
    8,n_intervals = 15,"gaussian")

# Find the solutions of optimization problem
solu <- findsolutions.lsm(vario, "exponential",
    c(30,46,0.5,100),upper.bound = c(46,46,pi/2,250))
```

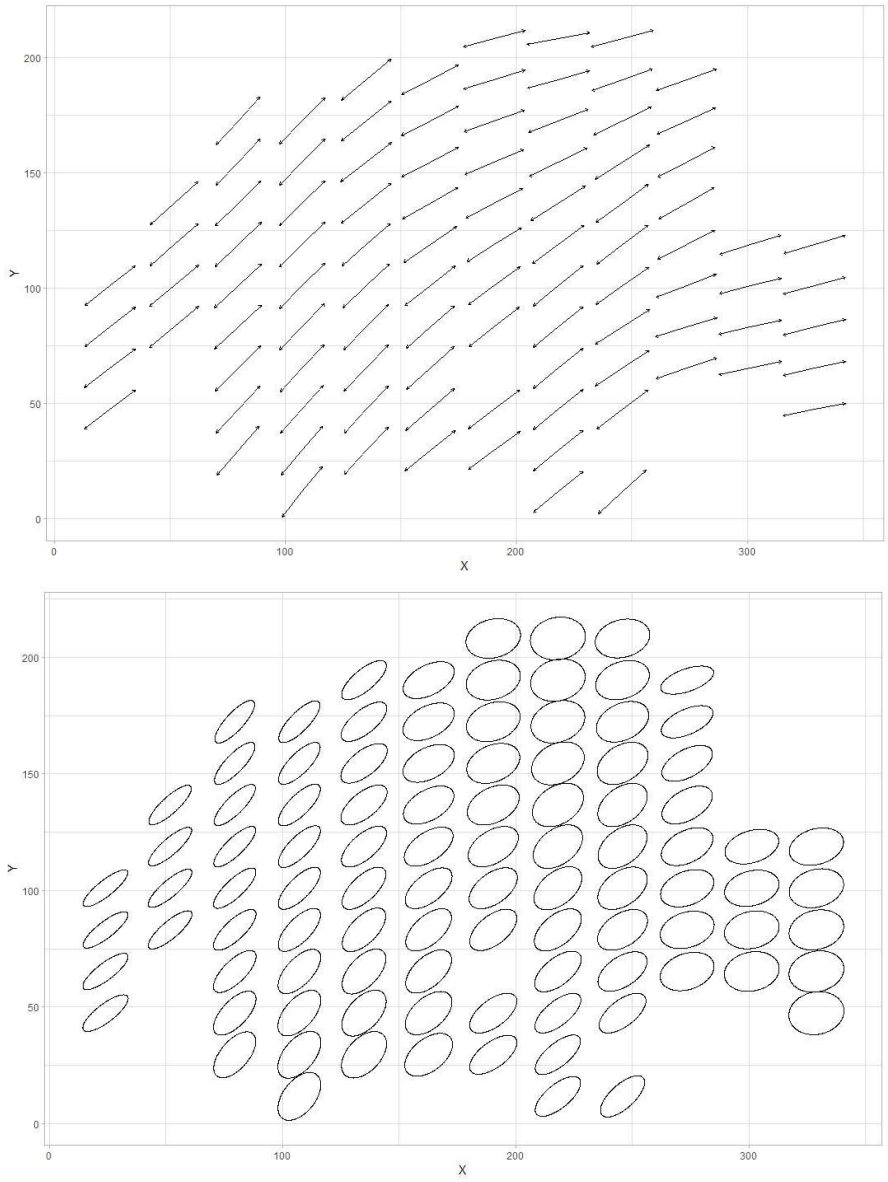


Figure 5.3: Anisotropy ellipses and azimuth

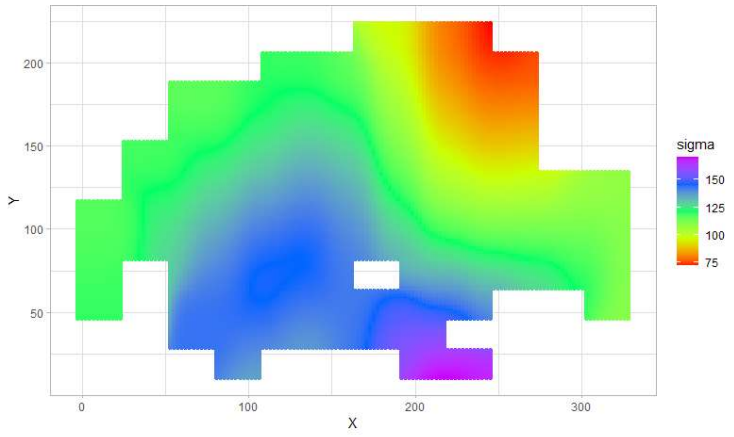


Figure 5.4: $\hat{\sigma}(\cdot)$

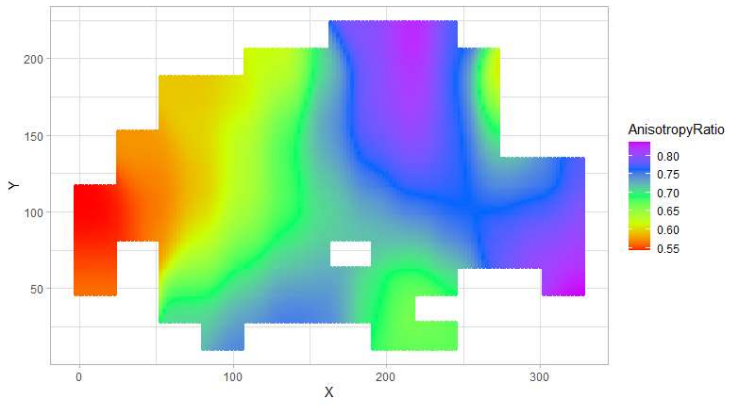


Figure 5.5: Estimated Anisotropy Ratio

We can see spatial patterns for anisotropy and variance are very similar to the plots reported in (1).

5.2 Simulated data

The following two examples are applications of this method to simulated non stationary random processes. The scripts used to simulate this type of processes are based on Stan and were provided by Dr. Scimone. Given a grid of spatial points, we need to define the parameters vector $\boldsymbol{\theta}_{\mathbf{s_0}} = \{\sigma_{\mathbf{s_0}}, \lambda_{1\mathbf{s_0}}, \lambda_{2\mathbf{s_0}}, \phi_{\mathbf{s_0}}, \mu_{\mathbf{s_0}}\}$ in each point of the grid and we get a realization of a random process with such mean and covariance structure. We want to see if, starting from a realization of such GP (gaussian process) we can retrieve the structure of the parameters vector. We report for 2 examples the "real" structure of parameters and the ones that were estimated by our package after some hyperparameter tuning with data simulated according to an exponential variogram structure.
For both simulations we have 3600 equally spaced points in a grid in $[-1;1] \times [-1;1]$.

5.2.1 Simulated example 1

For this first process we set both μ and σ as radially linearly increasing.

$$\mu(r) = 1 + r$$

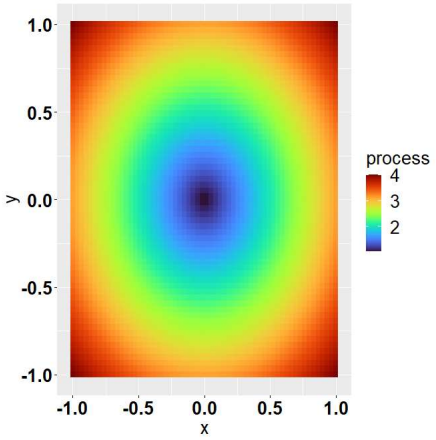
$$\sigma(r) = 1 + r$$

and both eigenvalues follow a "chessboard" pattern:

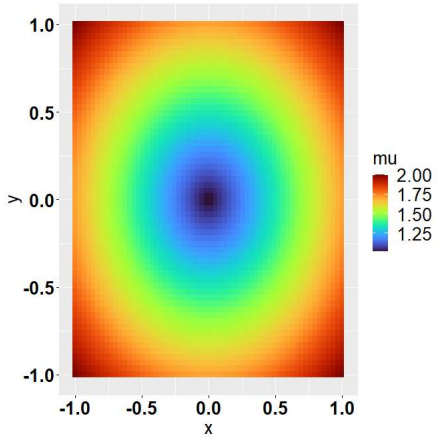
$$\lambda_1 = \sqrt{0.0005}\mathbb{1}_{\{xy \geq 0\}} + \sqrt{0.003}\mathbb{1}_{\{xy < 0\}}$$

$$\lambda_2 = \sqrt{0.003}\mathbb{1}_{\{xy \geq 0\}} + \sqrt{0.0005}\mathbb{1}_{\{xy < 0\}}$$

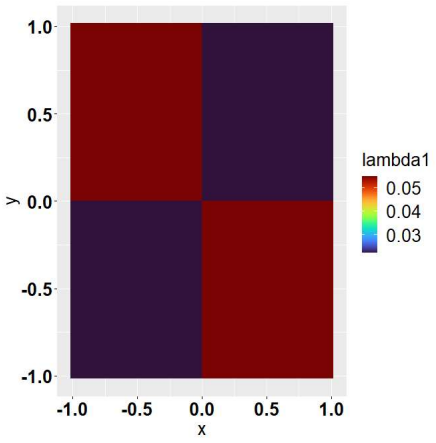
The angle parameter ϕ is set in such a way that one of the directions of anisotropy is always radial and by coupling eigenvalues and eigenvectors in this way, we get a continuous covariance function even though parameters are discontinuous.



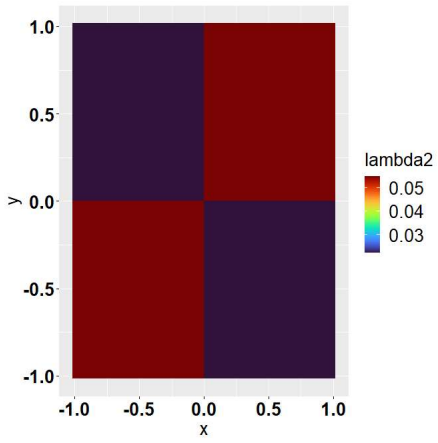
(a) σ



(b) μ



(c) λ_1



(d) λ_2

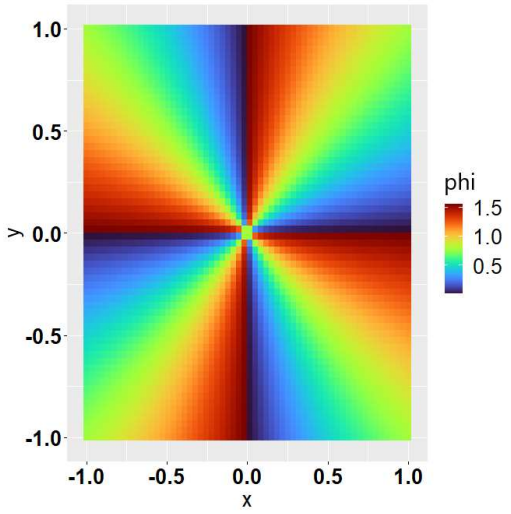


Figure 5.7: ϕ

We can see that this pattern is pretty well captured by the estimated parameters, and also σ is clearly radially increasing. The estimated angle is far from the real values; in general we observed that the method often fails to retrieve complex structures for the angle ϕ.

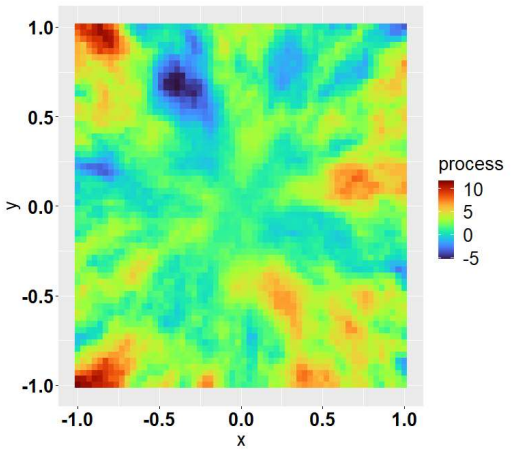


Figure 5.8: Non Stationary Spatial Random Process 1

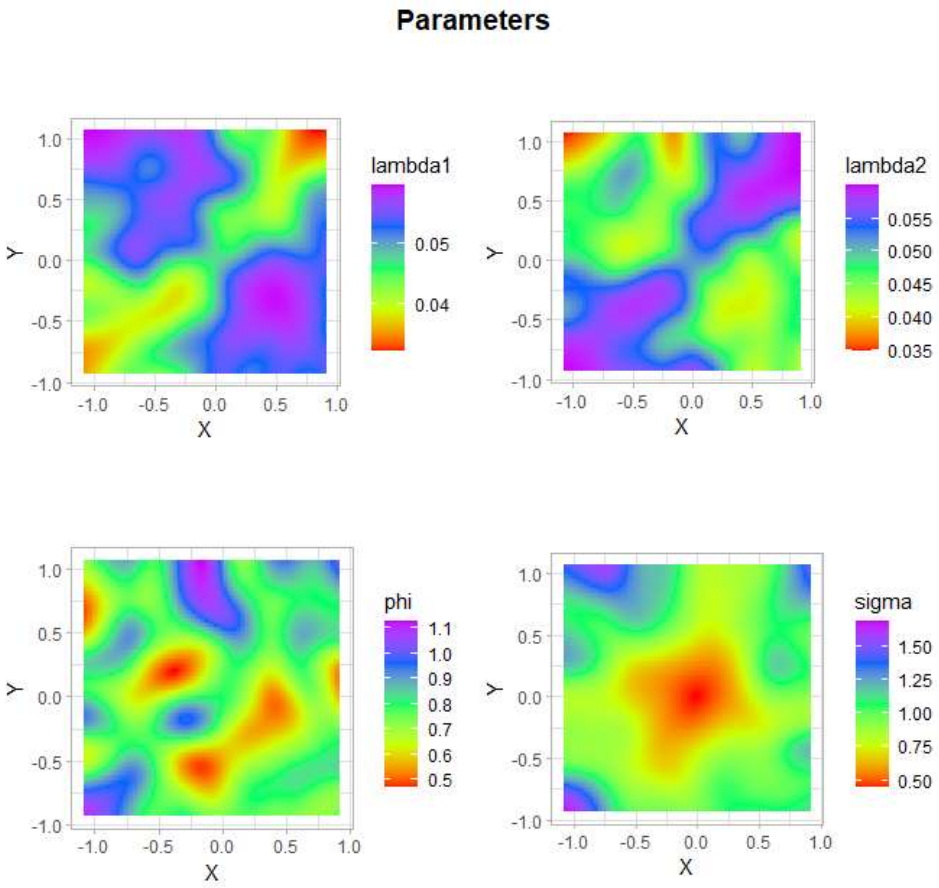


Figure 5.9: Estimated parameters in D

5.2.2 Simulated example 2

For this second example we have:

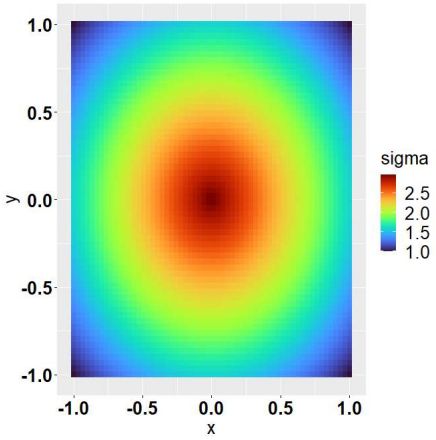
$$\mu(r) = 3 + r$$

$$\sigma(r) = 3 - 2r$$

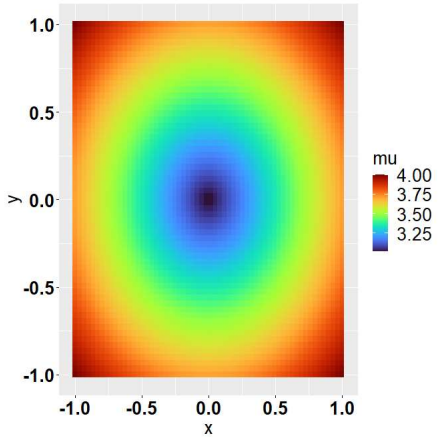
$$\lambda_1 = \sqrt{(0.004 + 0.0035y)}$$

$$\lambda_2 = \sqrt{(0.004 + 0.0035x)}$$

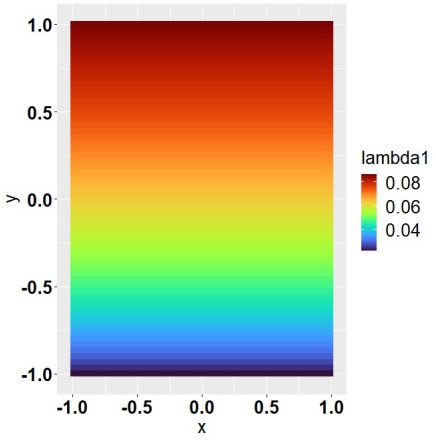
$$\phi = \frac{\pi}{4} \ \forall s \in D$$



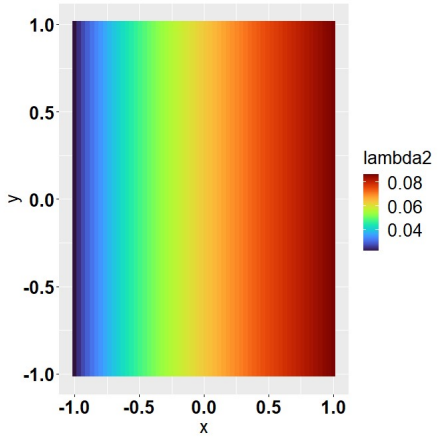
(a) σ



(b) μ



(c) λ_1



(d) λ_2

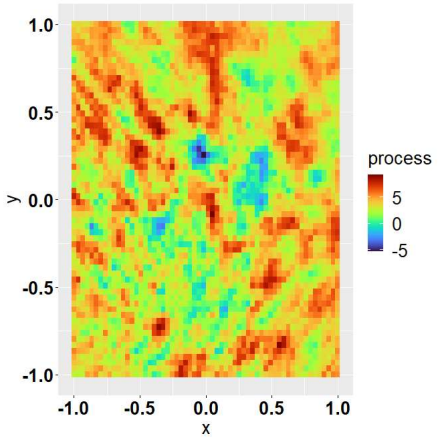


Figure 5.11: Non Staionary Spatial Random Process 2

The pattern for σ parameter is again well captured, and also the x-increasing and y-increasing structure for λ_1 and λ_2 respectively. It is clear that towards the center there is a greater loss for λ_1, λ_2 and ϕ; this can be easily justified because if $\lambda_1 \approx \lambda_2$, ϕ loses its meaning and any value of the angle yields the same results.

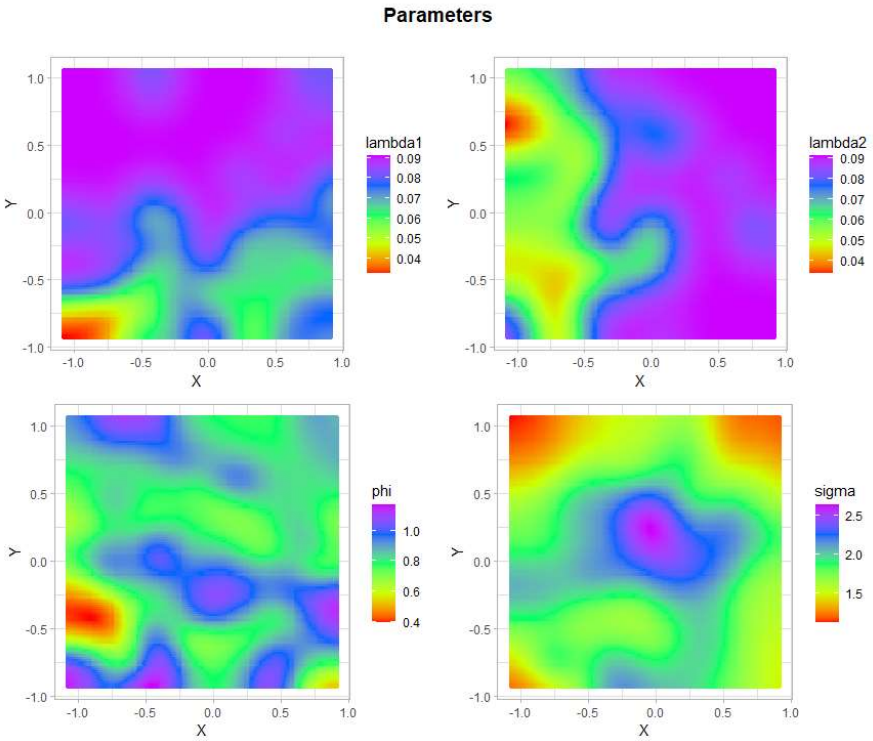


Figure 5.12: Estimated parameters in D

Chapter 6

HowTo

In this chapter we will have an overview of the R functions and see how to use our algorithm in a practical case by exploiting the dataset `meuse` from the `sp` package. The script `meuse.R` to run this example can be found inside the folder `test` of the source code. Since the dataset `meuse` is already provided by `LocallyStationaryModels`, it is not necessary to install also the `sp` package to run it.

6.1 Find the anchor points

First, we clean the environment, import the data and extract the matrix of the coordinates and the vector with the value of the elevation in every point.

```
# Load the data
data(meuse)
d <- cbind(meuse$x, meuse$y)
z <- meuse$elev
```

We can find the position of the anchor points calling the function `findanchorpoints.lsm`.

```
# Find anchorpoints
a <- find_anchorpoints.lsm(d, 12, TRUE)
```

We decided to create a 12×12 grid and let the package find the points. The object returned contains the position of them and the specifications of the grid. The anchor points found are less than 144, because most of the cells are empty. Since we passed TRUE as an argument, the functions provides a plot with the starting points in black and the anchor points as red crosses.

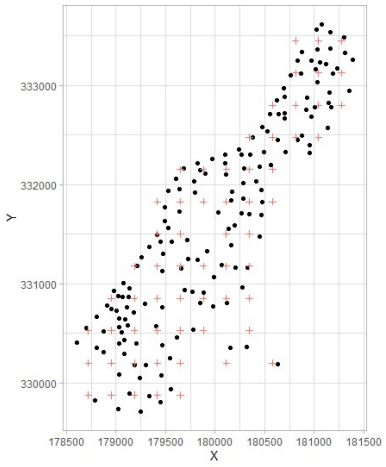


Figure 6.1: Anchor points in red

6.2 Calculate the sample variogram

We proceed calculating the sample variogram with the function `variogram.lsm`.

```
# Build the empiric variogram
vario <- variogram.lsm(z, d, a$anchorpoints, 370, 8, 8, "gaussian")
```

We opted for a gaussian kernel with $\varepsilon = 370$ and for a grid with 8 angles and 8 intervals for each angle. This function does not only calculate the variogram, but it also returns the kernel matrix, the grid matrix, vectors mean.x, mean.y and the squared weights.
We can plot the variogram in all 8 directions sequentially using the function `plotvario`.

```
plotvario(vario, 6)
```

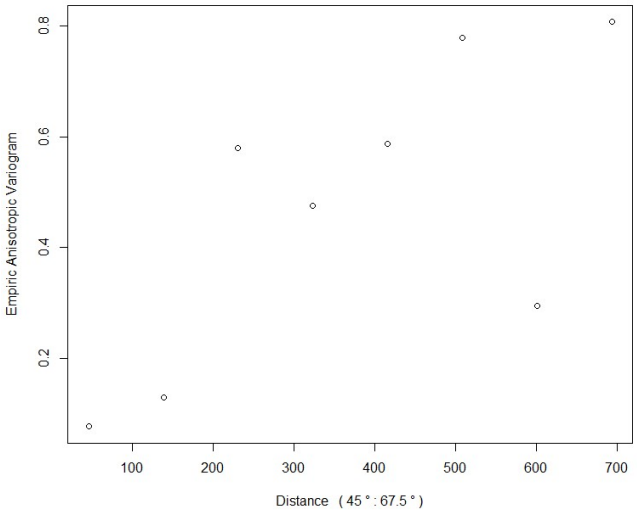


Figure 6.2: Empiric variogram in direction (45° : 67.5°)

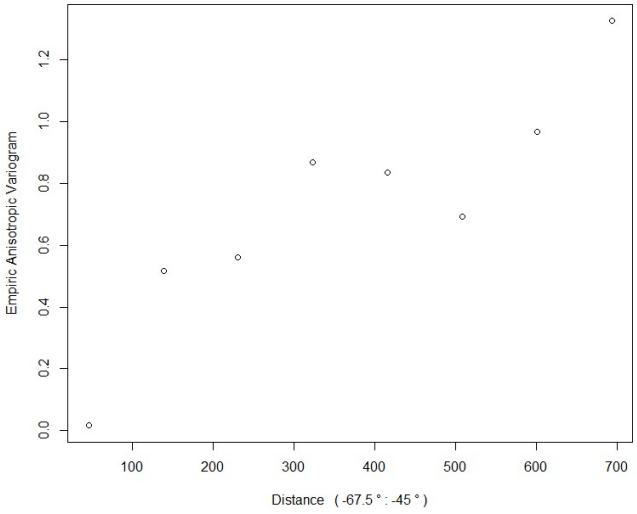


Figure 6.3: Empiric variogram in direction (-67.5° : - 45°)

6.3 Fit the sample variogram

We can fit the sample variogram with a variogram of our choice and find the optimal value of its parameters in each anchor point by calling the function `findsolutions.lsm`.

```
# Find the solutions
solu <- findsolutions.lsm(vario, "exponential", c(200,200,0.01,100))
```

We fitted an exponential variogram and set the initial parameter for the optimizer to $\lambda 1 = 200$, $\lambda 2 = 200$, $\phi = 0.01$ and $\sigma = 100$. We decided to leave the default values for the upper and lower bounds. As you can see the function takes everything it needs to compute the result from the object returned by `variogram.lsm` and provide not only the solutions of the non-linear optimization problem, but also the value of the optimal delta to perform smoothing and kriging on the other points of the plane.

6.4 Visualize the results

We can visualize the results of our analysis via `plot.lsm`.

```
# Plot of the solutions
mypoints<-plot.lsm(model = solu, a = a, z = y, d = d, n_points = 3,
    points_arrangement = "straight", kriging = FALSE,
                ellipse_scale = 2, arrow_scale = 1.5)
```

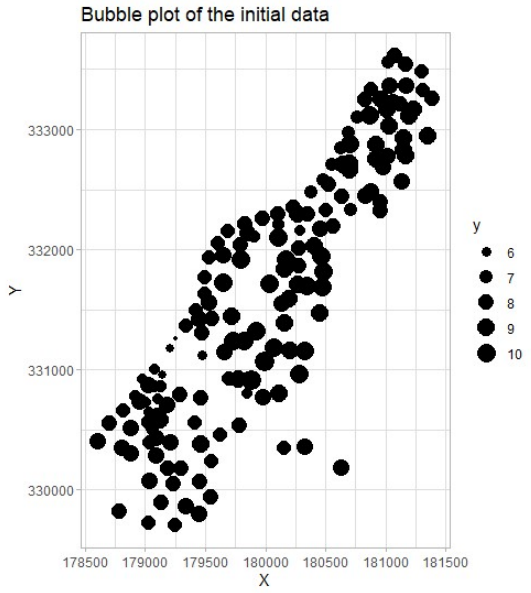


Figure 6.4: Bubbleplot of the initial data

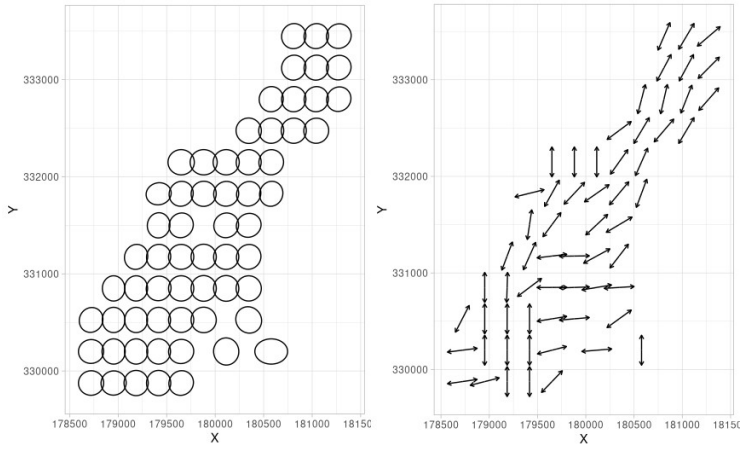


Figure 6.5: Anisotropy ellipses and directions

Given in input an object of class lsm (returned by the previous function), `plot.lsm` yields a bubble plot of the initial data, a pair of plots with the anisotropy ellipses and directions, a plot with the values of λ_1, λ_2, ϕ and σ in randomly generated points in the plane (if `points_arrangement` is equal to `"random"`) and two plots with the values of the mean and punctual value of z predicted in the same locations.

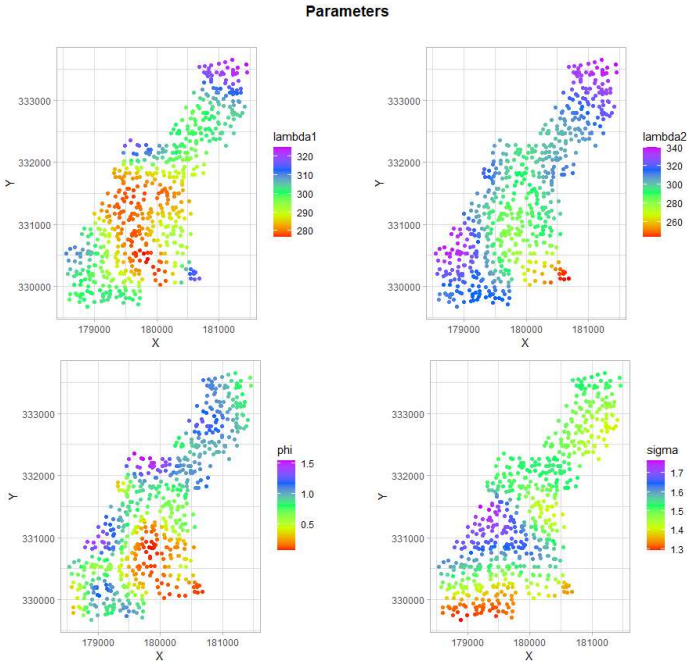


Figure 6.6: Plot of the parameters, $points\_arrangement = "random"$

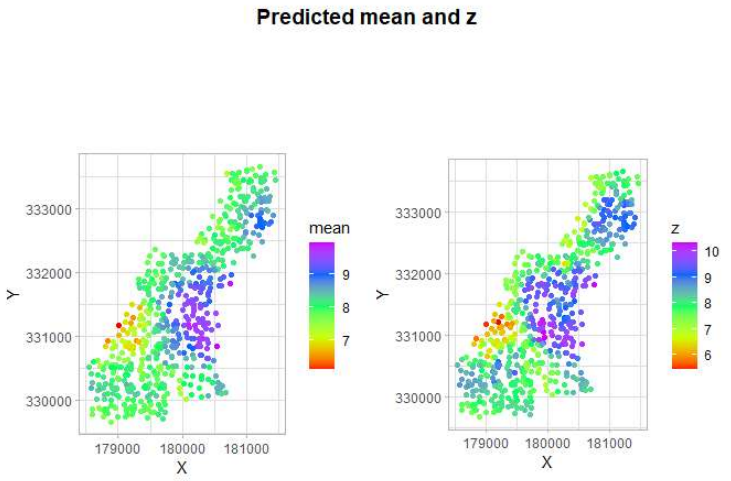


Figure 6.7: Kriging on random points

If we prefer to have equally spaced points instead of random generated ones we can set `points_arrangement` equal to `"straight"`.

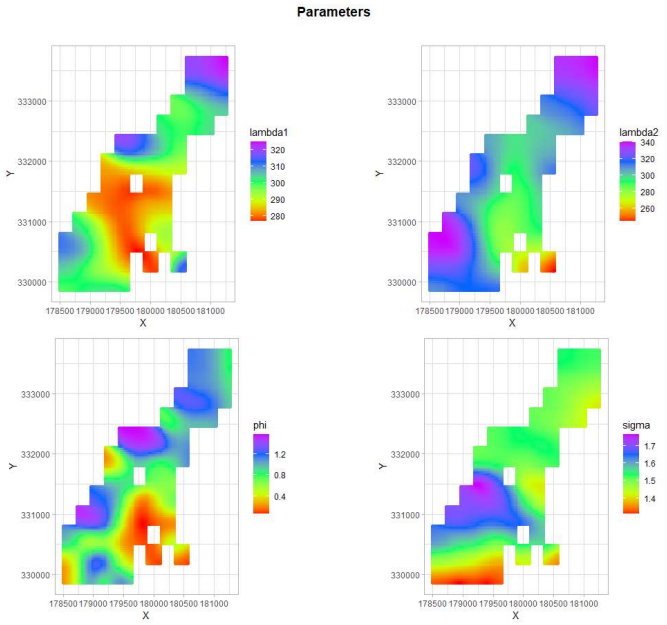


Figure 6.8: Plot of the parameters, $points\_arrangement = "straight"$

6.5 Kriging and error by cross-validation

Eventually we can perform kriging on the initial data and check that interpolates perfectly the data in z.

```
previsions <- predict.lsm(solu, d)
max(previsions$zpredicted - z)
```

The error our model can be estimated via cross-validation.

```
# Test the performace of our model via cross-validation
cv.lsm(z,d,a$anchorpoints,350,8,8,"gaussian","exponential",
    c(200,200,0.01,100))
```

Chapter 7

Installation

7.1 Install R and Rstudio (optional)

7.1.1 Windows

Download R from `https://cran.r-project.org/bin/windows/base/` and (optional) Rstudio from `https://www.rstudio.com/products/rstudio/download/`.

7.1.2 Arch Linux

Open the terminal then type:

```
sudo pacman -S r
```

and install the `r` package for R. To get Rstudio (optional), install an AUR helper of your choice such as `https://github.com/Jguer/yay` and then on the terminal:

```
yay -S rstudio-desktop-bin
```

Refer to `https://wiki.archlinux.org/title/r` for further details.

7.1.3 Ubuntu

Refer to `https://cran.r-project.org/bin/linux/ubuntu/` for R.
For Rstudio (optional) you can download the `.deb` file from `https://www.rstudio.com/products/rstudio/download/`.

7.2 Install LocallyStationaryModels

7.2.1 Via devtools (suggested)

Open R and type:

```
install.packages("devtools")
```

to install `devtools`. Then type:

```
library(devtools)
devtools::install_github("elucasticus/LocallyStationaryModels")
```

to install `LocallyStaionaryModels`.

7.2.2 Via zip file

In alternative you can download the source code from github, unzip the file and run from the terminal:

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

⚠**Beware:** since `LocallyStationaryModels` is written in C++, it requires the appropriate tools to be compiled and runned. On Windows this can be done by installing `https://cran.r-project.org/bin/windows/Rtools/rtools40.html` (don't worry, R itself will redirect you to the appropriate web page if `Rtools` is not present on your pc), while on Linux you have to manually install them from the repositories of your distribution.

Chapter 8

Future developments

The whole method and consequently our package was developed for scalar valued data. The local stationarity assumption and also the convolutional kernel method for variogram estimation can naturally be extended to vectorial and functional data. The code should be already well structured to handle the functional case and the estimation by means of trace-variography. While for vectorial data (both real and functional) some addition to the code might be needed, the structure should still be well suited for this task. Since local stationarity is mainly a theoretical framework and its definition is not very strict, we saw the method can work in contexts where covariance parameters even have discontinuities. Defining quasistationarity in functional context might pose some challenges, but maybe the weak characterization of this definition could make this method still feasible in practice.

Appendix A

Valid Variogram Models

We present some isotropic stationary variogram models implemented in our package.

Exponential model: **id** = "exponential"

$$\gamma(h) = \begin{cases} \sigma^2(1 - e^{-\frac{h}{a}}), & \text{if } h > 0 \\ 0, & h = 0 \end{cases} \tag{A.1}$$

a is set to 1 in the anisotropic setting to avoid overparametrization.

Gaussian model: **id** = "gaussian"

$$\gamma(h) = \begin{cases} \sigma^2(1 - e^{-\frac{(\frac{h}{r})^2}{a}}), & \text{if } h > 0 \\ 0, & h = 0 \end{cases} \tag{A.2}$$

r is set to 1 in the anisotropic setting to avoid overparametrization. The parameter a has different values in different references, due to the ambiguity in the definition of the range. E.g. $a = \frac{1}{3}$ is the value used in (ChilesDelfiner 1999) and also the default value in our package.

Matern model: **id** = "matern" or "maternNuFixed 12" (12 is an example for the fixed value of ν of choice)

$$\gamma_\nu(h) = \sigma^2\left(1 - \frac{2^{1-\nu}}{\Gamma(\nu)}\left(\sqrt{2\nu}\frac{h}{\rho}\right)^\nu K_\nu\left(\sqrt{2\nu}\frac{h}{\rho}\right)\right) \tag{A.3}$$

K_ν is the modified Bessel function of the second kind, Γ is the gamma function. A Gaussian process with Matérn covariance is $\lceil \nu \rceil - 1$ times differentiable in the mean-square sense.
in the package parameter ν can either be learnt through the optimization problem as all other parameters or it can be fixed a priori by choosing model "maternNuFixed nu" with "nu" substituted by the chosen ν parameter.

ρ is set to 1 in the anisotropic setting to avoid overparametrization. It is a well known result that the exponential model is a specific case of a Matérn variogram with $\nu = \frac{1}{2}$. The gaussian covariance is the limit for $\nu \to \infty$ of the Matérn covariance.

$$\lim_{\nu \to \infty} \gamma_\nu(h) = \sigma^2(1 - e^{-\frac{h^2}{2r^2}})$$

Bibliography

[1] Francky Fouedjio, Nicolas Desassis, Jacques Rivoirard, (2016). A generalized convolution model and estimation for non-stationary random functions. Spatial Statistics. .

[2] Matheron, G.F., (1971). The Theory of Regionalized Variables and its Applications. Les Cahiers du Centre de Morhologie Mathématique de Fontainebleau, vol. 5. Ecole Nationale Supérieure des Mines de Paris.

[3] Wand, M., Jones, C., (1995). Kernel Smoothing. Monographs on Statistics and Applied Probability, Chapman and Hall.

[4] Menafoglio A., Geostatistical analysis of spatially dependent data: from real to Hilbert-space valued random fields.

[5] Paciorek, C.J., Schervish, M.J., (2006). Spatial modelling using a new class of nonstationary covariance functions. Environmetrics 17, 483–506.

[6] Writing R Extensions. https://cran.r-project.org/doc/manuals/R-exts. pdf, 2018.

[7] D. Eddelbuettel, R. Francois, (2011). Rcpp: Seamless R and C++ Integration, Journal of Statistical Software, 40(8): 1 – 18.

[8] Cressie, Noel (1993). Statistics for Spatial Data.

[9] Byrd, R. H.; Lu, P., Nocedal, J., Zhu, C. (1995). A Limited Memory Algorithm for Bound Constrained Optimization. SIAM J. Sci. Comput.