POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

ProbKMA: Optimization of the Probabilistic K-means with Local Alignment Algorithm

Advanced Programming for Scientific Computing Project
Mathematical Engineering

Niccolò Feresini, Riccardo Lazzarini

Supervisor:
Prof. Marzia A. Cremona
Academic year:
2023-2024

Abstract: This project deals with functional data analysis. In particular, we efficiently implement an algorithm for locally clustering curves and discovering functional motifs. The algorithm, developed by Marzia A. Cremona and Francesca Chiaramonte (2020) [1], had a previous R implementation. However, this implementation lacked efficiency, especially when dealing with large datasets. We provide two alternative implementations: one commissioned by Professor Marzia A. Cremona, focused on the rewriting of some of the most computationally expensive parts, leaving parts in `R` for forthcoming extensions by programmers not skilled in the `C++` language, and a second consisting of a complete implementation in `C++` of the core of the algorithm, namely the ProbKMA function. The source code for the project is available at `https://github.com/NiccoloF/ProbKMA-FMD` in the branches named main_1,main_3. At the end of the report, we provide instructions on package installation and running examples.

Key-words: Functional Data Analysis, K-means, Scientific Computing, Functional Motif Discovery

1. Introduction

Unsupervised learning methods play a crucial role in statistics, revealing hidden patterns and relationships in data without specific guidance. Their importance spans various applications, providing valuable insights into complex datasets and enabling systematic, data-driven exploration. Particularly noteworthy is their relevance in Functional Data Analysis (FDA). Given the challenges of working with intricate, frequently misaligned, and high-dimensional functional data, the capabilities of unsupervised learning become essential for extracting meaningful information within the FDA framework.

1.1. Functional Motif Discovery

In this project, we consider the task of discovering functional motifs within a set of curves, the functional data, aiming to identify typical shapes recurring within each curve and across several curves in the set. This problem finds applications in various scientific domains, including bioinformatics, genomics, finance, and engineering, where the goal is to recognize conserved segments or patterns within biological structures, stock prices, or sensory time series.

Specifically, we explore the unsupervised method proposed by [1] to address the functional motif discovery task. This method called *probabilistic K-mean with local alignment* (probKMA), introduces an innovative approach that utilizes local curve alignment to identify shared segments without the need for prior knowledge about motif characteristics, such as the number, lengths, or radii. Furthermore, the lengths and radii are customized for each motif, eliminating the reliance on predetermined values.

Following this approach, clusters are locally defined on segments of misaligned curves, allowing each cluster to include multiple segments of the same curve. Since this method falls within the realm of Functional Data Analysis (FDA), it enables the incorporation of derivatives in the dissimilarities definition, it relies on a rigorous notion of variability within each motif and on noise reduction in the curves through smoothing. Drawing inspiration from the FDA literature, this method resembles the K-mean with global alignment [6] by performing clustering and alignment but opts for local alignment. Similar to bioinformatics practices, the algorithm extends the domain of clusters using high-similarity "seeds". Lastly, akin to fuzzy clustering, the curves can be associated with an arbitrary number of clusters, deviating from the usual restriction to only one cluster as in traditional clustering methods.

1.2. Our contribute

We developed two `R` packages in which we rewrite part of the code using the `C++` programming language. In the first package, certain functionalities, like the ProbKMA algorithm's structure, motif updating, or distance calculation, remain in `R`. This design choice ensures that any future modifications or expansions in the `R` code, such as introducing new distances, will not necessitate changes in the `C++` code. In the second package, we completely rewrite the core of the algorithm, the ProbKMA function. This allows for more flexibility in code design and the selection of data structures. Once again, the focus is on code generality and potential future expansions, but implementing these would require `C++`. Both packages offer improved efficiency compared to the previous `R` implementation, enabling analyses on large datasets with numerous missing values that would have previously required lengthy computation times.

1.3. Report structuree

This report has the following structure:

- in Section 2, we briefly explain the mathematical background behind the algorithm and the main parts of which it is composed;
- in Section 3, we explain how to interface `R` and `C++` and how an `R` package that also contains pre-compiled code is structured;
- in Section 4, we describe the code interventions made in the first package, the data structures and the public interface for using it;
- in Section 5, we describe the second package and, in particular, the code structure, the implemented classes, the choice of data structures, the user interface and the parallelised algorithm sections;
- in Section 6, we show practical examples of code application and comparisons of computational times for various implementations;
- in Section 8, we answer some practical questions: how to extend our code in the future and how to install the package.

2. The Algorithm

2.1. Mathematical Setting

In the context of functional data analysis, the dataset consists of N (d-dimensional) curves, i.e. $\mathbf{x}_i : \mathbb{R} \to \mathbb{R}^d$, $i = 1, ..., N$. The goal of the ProbKMA algorithm is to obtain K (d-dimensional) cluster centers $\mathbf{v}_k : (0, c_k) \to \mathbb{R}^d$, $k = 1, ..., K$ that are "patterns" to which the curves exhibit local high similarity, as measured by the distance $d(\cdot, \cdot)$.

The clusters lengths c_k depend on k and are contained in the interval $[c_{\min}, c_{\max}]$. Given the focus on local similarity between segments of curves, and the definition of each cluster center exclusively within the interval $(0, c_k)$, we enable the alignment of each curve with each cluster center to minimize their distance. Alignment is performed composing each curve $\mathbf{x}_i$ with a warping function $h_{k,i} : \mathbb{R} \to \mathbb{R}$, taken from the class $W := \{h : t \mapsto t + s; s \in \mathbb{R}\}$, i.e. the class of all possible shifts.

Concerning the distance $d(\cdot, \cdot)$, until now, a Sobolev-type distance of the following form has been considered:

$$d_\alpha(\mathbf{x}, \mathbf{v}) = \left(\sum_{j=1}^{d} \frac{w_j}{d} \left[\frac{1-\alpha}{c} \int_0^c (x^{(j)}(t) - v^{(j)}(t))^2 dt + \frac{\alpha}{c} \int_0^c (x'^{(j)}(t) - v'^{(j)}(t))^2 dt \right] \right)^{1/2}, \tag{1}$$

where c denotes the length of the cluster $\mathbf{v}$, $\alpha \in [0, 1]$ is a fixed parameter and $w_j > 0$ is the weight of the j-th component of a d-dimensional curve, indicated by the superscript (j). In the following we will denote by $\tilde{\mathbf{x}}_{i,s_{k,i}} := \mathbf{x}_i \circ h_{k,i}$ the shifted curve. We recall that $\mathbf{x}'$ denotes the weak derivative of $\mathbf{x}$. The case $\alpha = 0$ leads to an L^2-like pseudo-distance involving only the curves, while the opposite case $\alpha = 1$ involves only the weak derivative information. When $\alpha \in (0, 1)$, we are considering a H^1-like pseudo distance, which involves both levels and variations of the curves.

Due to the emphasis on local similarity, a curve can be associated with multiple clusters, allowing different curve segments to exhibit similarity to segments of other curves. Following a fuzzy clustering approach, probabilities $p_{k,i}$ are assigned to each curve $\mathbf{x}_i$, indicating its membership in each cluster k. For each $k \in \{1, ..., K\}$ we define the membership function $p_k : \{\mathbf{x}_1, ..., \mathbf{x}_N\} \to [0, 1]$ with $p_k(\mathbf{x}_i) = p_{k,i}$ such that $\sum_{k=1}^K p_{k,i} = 1$ for all $i = 1, ..., N$ and $\sum_{i=1}^N p_{k,i} > 0$ for all $k = 1, ..., K$. Each membership probability corresponds to a specific shift $s_{k,i}$ of the curve $\mathbf{x}_i$, minimizing the distance between $\mathbf{x}_i$ and the corresponding cluster center $\mathbf{v}_k$. These shifts are summarized in a matrix $S = [s_{k,i}] \in \mathbb{R}^{K \times N}$, and the membership probabilities are captured in a matrix $P = [p_{k,i}] \in \mathbb{R}^{K \times N}$.

2.2. Optimization Problem

Let us consider the lengths of the cluster centers fixed. ProbKMA can be formulated as the following optimization problem (ProbKMA-OP):

Find $\mathbf{v}_1, ..., \mathbf{v}_K$, P and S that minimize the following functional:

$$J_m(P, S, \mathbf{v}_1, ..., \mathbf{v}_K) = \sum_{i=1}^{N} \sum_{k=1}^{K} (p_{k,i})^m \; d_\alpha^2(\mathbf{x}_i \circ h_{k,i}, \mathbf{v}_k) \tag{2}$$

under the constraints $p_{k,i} \in [0, 1]$ for all i, k such that $\sum_{k=1}^K p_{k,i} = 1, \; \forall i$ and $\sum_{i=1}^N p_{k,i} > 0, \; \forall k$. Here $m > 1$ denotes a fixed weighting parameter.

To solve this optimization problem, we rely on an iterative procedure that alternates two steps:

1. given $\hat{S}$ and K cluster centers $\hat{\mathbf{v}}_1, ..., \hat{\mathbf{v}}_k$ computed at the previous iterations, update the membership probabilities according to the following rule:

$$\hat{p}_{k,i} = \left[\sum_{l=1}^{k} \left(\frac{d_\alpha^2(\tilde{\mathbf{x}}_{i,\hat{s}_{k,i}}, \hat{\mathbf{v}}_k)}{d_\alpha^2(\tilde{\mathbf{x}}_{i,\hat{s}_{k,l}}, \hat{\mathbf{v}}_l)} \right)^{1/(m-1)} \right]^{-1} \qquad k = 1, ..., K \tag{3}$$

for all $i \in R := \{i \in \{1, ..., N\} : d_\alpha^2(\tilde{\mathbf{x}}_{i,\hat{s}_{k,i}}, \hat{\mathbf{v}}_k) > 0 \text{ for all } k\}$ and

$$\hat{p}_{k,i} = \begin{cases} 0, & k \; : \; d_\alpha^2(\tilde{\mathbf{x}}_{i,\hat{s}_{k,i}}, \hat{\mathbf{v}}_k) > 0 \\ \in [0, 1], & k \; : \; d_\alpha^2(\tilde{\mathbf{x}}_{i,\hat{s}_{k,i}}, \hat{\mathbf{v}}_k) = 0 \end{cases} \tag{4}$$

with $\sum_{k=1}^K \hat{p}_{k,i} = 1$, for all $i \notin R$;

2. given the membership matrix $\hat{P}$ and a shift matrix $\hat{S}$, update the cluster centers according to the following formula:

$$\hat{\mathbf{v}}_k = \frac{\sum_{i=1}^N (\hat{p}_{k,i})^m \; \tilde{\mathbf{x}}_{i,\hat{s}_{k,i}}}{\sum_{i=1}^N (\hat{p}_{k,i})^m}, \quad \text{a.e. in } (0, c_k), \forall \; k. \tag{5}$$

For $\alpha = 1$, $\hat{\mathbf{v}}_k$ is defined by (5) up to an additive constant.

2.3. ProbKMA

The algorithm proposed by Cremona and Chiaromonte (2020) to solve ProbKMA-OP is the following:

Inizialization Fix K, $c_1, ..., c_K$ and consider an initial membership matrix $P^{(0)}$ and shift matrix $S^{(0)}$;
Iteration for it $= 1, 2, ...$ iterate until convergence:

1. *Identification of cluster centers*: Using $s_{k,i}^{(\text{it}-1)}$ and $p_{k,i}^{(\text{it}-1)}$, compute the k-th cluster center $\mathbf{v}_k^{(\text{it})}$ with equation (5);
2. *Curve alignment*: For all i, k, find the shift $s_{k,i}^{(\text{it})}$ that applied to the curve $\mathbf{x}_i$ minimizes the distance $d_\alpha(\tilde{\mathbf{x}}_{i,s}, \mathbf{v}_k^{(\text{it})})$;
3. *Computation of membership probabilities*: Using $\mathbf{v}_k^{(\text{it})}$ and $s_{k,i}^{(\text{it})}$, update the membership matrix $P^{(\text{it})}$ using equations (3) and (4).

Stopping criterion The algorithm converges when the global Bhattacharyya distance (GBD) goes under a given tolerance. In particular, GDB is computed as the maximum, mean, or order q quantile of the following vector of distances involving the membership matrices $P^{(\text{it})}$ and $P^{(\text{it}-1)}$:

$$BC_i = -\log\left(\sum_{k=1}^{K}\sqrt{p_{k,i}^{(\text{it})}p_{k,i}^{(\text{it}-1)}}\right), \quad i = 1, ..., N. \tag{6}$$

In paper [1] , it is shown that along the iterations of the algorithm, the functional J_m decreases. As the authors point out, this property is only a necessary condition to converge to a global minimizer of J_m.

2.4. Cluster Lengths Selection

In the previous algorithm, the lengths $c_1, ..., c_K$ remain fixed. Generally, these quantities are unknown a priori, and we want a procedure to identify them. The approach proposed by [1] consists of computing short cluster centers and using them as seeds. These seeds are extended to both sizes during an additional step of the above algorithm, called *center elongation*, performed when the algorithm is close to convergence. The elongation is only accepted if it leads to an objective function smaller or higher than a given threshold $\Delta_{J_{m,k}}$ to the one before the elongation.

2.5. Cluster cleaning

ProbKMA faces challenges in distinguishing between a curve $\mathbf{x}_{i_1}$ that closely aligns with all K cluster centers and another curve $\mathbf{x}_{i_2}$ that doesn't match any of the K cluster centers. The issue arises as both cases result in identical membership probabilities. We introduce a *cluster cleaning* step when the algorithm is near convergence to solve this problem, where the membership matrix P is dichotomized, converting probabilities to 0 or 1. We consider the quantile $q_{1/K}$ of order $1/K$ of all distances $d_\alpha(\mathbf{x}_i \circ h_{k,i}, \mathbf{v}_k)$; membership probabilities $p_{k,i}$ corresponding to distances lower than $q_{1/K}$ are set to 1, while all others are set to 0. This cleaning step differentiates extreme cases, leading to clean membership probabilities for each curve ($p_{k,i_1} = 1$ and $p_{k,i_2} = 0$, $k = 1, ..., K$). This process is repeated at the end of the iterations to identify curve portions belonging to each cluster, enhancing cluster center estimates. The refined outcomes are crucial for computing the generalized silhouette index and post-processing in functional motif discovery, explained in Section 2.8.

2.6. Handling of missing values

ProbKMA works best with reasonably smooth curves. In real applications, each functional datum must be created from discrete evaluations, possibly available on datum-specific and/or irregular grids, with some measurements missing relative to other data. Smoothing and other straightforward pre-processing steps can handle this problem by filling small gaps. However, when input curves present large gaps, these missing subregions cannot be imputed by smoothing. Functional methods that consider the curves globally are not suitable for this kind of data. On the contrary, ProbKMA can tolerate large gaps because it exploits the functional data locally.

We allow each input curve $\mathbf{x}_i$ to be defined in a domain $D_i \subseteq \mathbb{R}$ consisting of a finite union of intervals. In this case, the distance is generalized in (1) as follows:

$$\tilde{d}^2_\alpha(\mathbf{x}, \mathbf{v}) = \sum_{\nu=1}^{d} \frac{w_\nu}{d} \left[\frac{1-\alpha}{|(0,c) \cap D|} \int_{(0,c)\cap D} (x^{(\nu)}(t) - v^{(\nu)}(t))^2 dt + \frac{\alpha}{|(0,c) \cap D|} \int_{(0,c)\cap D} (x'^{(\nu)}(t) - v'^{(\nu)}(t))^2 dt \right] \quad (7)$$

where D denotes the domain of the curve $\mathbf{x}$. Based on (7), the formula (5) for updating the cluster center becomes:

$$\hat{\mathbf{v}}_k = \frac{\sum_{i=1}^{N} \frac{(\hat{p}_{k,i})^m}{|(0,c_k) \cap \tilde{D}_{i,s_{k,i}}|} 1_{(0,c_k)\cap \tilde{D}_{i,\hat{s}_{k,i}}} \tilde{\mathbf{x}}_{i,\hat{s}_{i,k_i}}}{\sum_{i=1}^{N} \frac{(\hat{p}_{k,i})^m}{|(0,c_k) \cap \tilde{D}_{i,s_{k,i}}|} 1_{(0,c_k)\cap \tilde{D}_{i,\hat{s}_{k,i}}}} \quad (8)$$

a.e. on $(0, c_k) \cap \left(\cup_{i=1}^{N} \tilde{D}_{i,\hat{s}_{k,i}} \right)$. Here, 1_A denotes the indicator function of the set A and $\tilde{D}_{i,s} = D_i - s$ the domain of the shifted curve. In the case of $\alpha = 1$, $\hat{\mathbf{v}}_k$ is defined up to an additive constant.

2.7. ProbKMA Silhouette

To assess the clustering results generated by probKMA, the work [1] introduces a generalized silhouette index inspired by the one proposed by Rousseeuw (1987) [5] used in the classic clustering approach. This index, tailored for portions of curves, evaluates how well each segment fits with its assigned cluster.

Initially, we segment each curve $\mathbf{x}_i$ into portions corresponding to the clusters, achieved by binarizing the membership probabilities P into 0 and 1. Subsequently, we calculate the average distance $d_j(k)$ for each extracted portion $j = 1, ..., J$, representing the mean distance between j and all portions allocated to cluster k.

The intra-cluster distance a_j is defined as the average distance of portion j from its assigned cluster k_j (i.e., $a_j = d_j(k_j)$), while the inter-cluster distance b_j is the minimum of the average distances of portion j from all other clusters (i.e., $b_j = min_{k \neq k_j} d_j(k)$). The generalized silhouette index for portion j, denoted as s_j, is a value in $[-1, 1]$ and is calculated as $(b_j - a_j)/\max(b_j, a_j)$.

Higher s_j values signify appropriate cluster assignments for portion j, while lower values indicate suboptimal assignments. Negative values suggest that portion j is closer to a cluster different from the assigned one. For each cluster k, we compute the cluster average silhouette index S_k, representing the average silhouette index across all portions assigned to k. Finally, the overall average silhouette index S, obtained by averaging all S_k values, provides a comprehensive measure of the clustering quality.

2.8. Post-processing

ProbKMA, akin to other K-means algorithms, identifies a local minimum of the functional J_m, and its output is sensitive to the initialization. For *clustering curves* into K groups based on local similarity, multiple runs of the algorithm with diverse initializations (potentially with varied initial lengths) are performed, and the solution with the lowest J_m value is selected. In the case of *functional motif discovery*, the algorithm is executed several times with different initializations, cluster numbers, and motif initial lengths. The set of candidate motifs is formed by taking the union of the resulting solutions. This set is refined by employing generalized silhouette indices and considering motif occurrences. Similar candidate motifs are merged, as they may represent the same "true motif" identified in multiple runs of ProbKMA. Subsequently, a *motif search algorithm* is employed to locate all instances of the discovered motifs in the input curves.

2.9. Functional motif discovery algorithm

The post-processing stage of ProbKMA involves merging similar candidate motifs and identifying instances of the final set of functional motifs. Multiple strategies can be employed for merging and motif search, each with advantages and disadvantages.

To summarise the information from multiple ProbKMA runs, the proposed implementation groups similar motifs based on pairwise distances, selecting a representative motif in each group. The selection considers the motif's occurrences, average distance to occurrences, and length. This process is performed using hierarchical clustering, as follows:

1. Compute pairwise distances between candidate motifs;
2. Perform hierarchical clustering of motifs using average linkage;
3. Determine a global radius, denoted by R_{all}, based on minimum distances between all candidate motifs and curves;
4. Cut the hierarchical clustering dendrogram at a height of $2R_{\text{all}}$, creating M groups of similar motifs;
5. For each group $m = 1, ..., M$:
   (a) Determine a group-specific radius R_m, based on minimum distances between motifs in the group and all curves;
   (b) For each motif in the group:
      i. Identify curves containing the motif, i.e. with distance $\leq R_m$;
      ii. Approximate the number of occurrences and the average within-motif distance.
   (c) Select a small number of motifs based on occurrences, average distance, and length;
   (d) Find all occurrences of the selected motifs, i.e. portions of curves with distance $\leq R_m$ from the motif (*motif search step*).

The selection of the global radius R_{all} and group-specific radii R_m leverages information from distances between motifs and curves, considering the variability within each group. The method employs k-nearest neighbours to define radii based on discriminative distances between motifs and curves containing or not the motif.

Motif selection within each group is based on maximizing the approximate number of occurrences while minimizing the approximate average within-motif distance. The top-ranked motif in each group is usually selected, with an option to include motifs with significantly different lengths. Finally, a motif search step ensures accurate identification of occurrences, considering separation criteria to avoid counting the same motif occurrence multiple times.

3. Implementation

Creating an efficient `R` package for local alignment and functional motif discovery involves addressing three key challenges:

- Integrating `R` with `C++`;
- Choosing appropriate data structures for functional data storage;
- Implementing code parallelization;

This section elaborates on proposed solutions for these challenges in both implementations. Moreover, we describe some programming and code design techniques used.

3.1. `R` / `C++` interface

The communication interface between `R` and the `C++` source code is built using the *Rcpp package*, as detailed in Dirk and Romain [2]. Rcpp serves as a potent tool for seamless integration, enabling the incorporation of precompiled code within `R` to achieve enhanced computational performance. Its utility stems from its capability to simplify the embedding of `C++` code in `R`. In this way, developers can rely on the efficiency of `C++` algorithms while retaining the high-level abstractions and user-friendly features inherent in `R`.

To export classes and functions from `C++` to `R`, Rcpp employs a set of macros and conventions. For instance, the `Rcpp::export` attribute facilitates exposing `C++` functions to the `R` environment while the function `.Call` provides a powerful, yet simple, way to use precompiled code in `R`. Additionally, Rcpp utilizes the `Rcpp module` to encapsulate `C++` classes and methods.

Finally, in our project, we rely on *RcppArmadillo* [3], an extension of Rcpp that provides an interface to the *Armadillo library*, which is a high-quality linear algebra library for `C++`. It offers a similar syntax and functionality to `Matlab` and supports various matrix operations, statistics, and more. This library provides efficient classes for matrices and vectors as well as more complex data structures suitable for handling functional data.

3.2. Package Structure

Both packages are structured as follows:

- **R directory**: containing `R` code files (`.R`), as, for instance, functions and scripts for the user interface;
- **src directory**: containing `C++` source code files (`.cpp`), along with the *Makevars* files, which specify the flags needed for compiling and linking;
- **include directory**: containing the header files of the `C++` code (`.h`);
- **man directory**: containing documentation files (`.Rd`) for the `R` functions of the package. It is automatically generated using the package *Roxygen2* [7];
- **DESCRIPTION file**: it provides metadata about the package and includes information like package name, version, dependencies, author and description;
- **NAMESPACE file**: containing namespace directives to manage the function exports to the `R` environment.
- **data directory**: containing `.RData` files, i.e. the data included in the package;
- **tests directory**: containing the tests developed using the *testthat* library.

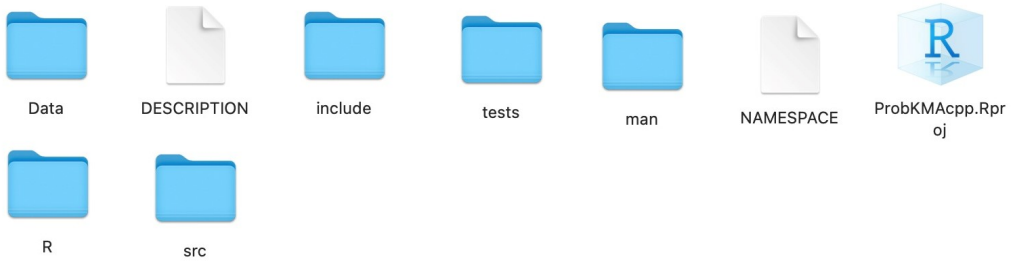

Figure 1: Package directory.

4. First Implementation

The first package we implemented includes the most computationally expensive functions and parts of the previous `R` version rewritten in `C++`. We design this package to make possible future code extensions, such as new distance types inclusion and the related calculation of motifs, so that these do not require changes to the `C++` code but only to the `R` code. In particular, the functions `R`: `diss_d0_d1_L2`, which computes the distances between two functional data, and `compute_motifs`, which updates the motifs found by probKMA, have been left in `R` and passed as objects of class `Rcpp::Function` to the `C++` code. This entails a limitation concerning the choice of data structures that we used to handle the functional data. In the following, we briefly describe our interventions in the code, justifying the choices and problems related to this first implementation.

4.1. Data Structures

The probKMA algorithm takes N curves and their derivatives as input. In practice, the functional data used constitutes an evaluation in discrete grids of each curve, which implies that each curve with values in $\mathbb{R}^d$ corresponds to a matrix with a certain number of rows, the length of the curve, and d columns. The same applies to its derivatives.

The algorithm allows each curve to have its length. For this reason, the previous `R` implementation used as data structure an R list of R lists containing two R matrices, one for the curve and one for its derivative and not a 3-dimensional array. In this way, the distance calculation was done by taking as input a list containing two matrices for the appropriately shifted curves and a list containing two matrices for the cluster center.

Given the use of some functions in their previous version `R` and the partial rewriting of some parts of the code, the data structure used to handle the functional data and motifs in this implementation is a `Rcpp::List` of `Rcpp::List` that contains two matrices `arma::mat` from the *Armadillo* library. In this way, Rcpp automatically handles the data structure transition from `R` to `C++` without having to do any cast. The problem is that using `Rcpp::List` requires a higher computational cost to create and access the data structures. Moreover, `Rcpp::List` structures are not *multi-threads safe*, and it is not possible to parallelize the `C++` code using *OpenMP*.

4.2. Code interventions

In this first implementation, we wrote in `C++` the *find shift warping minimizing dissimilarities* section within the ProbKMA algorithm, which in the code corresponds to the functions `find_shift_warp_min` and `find_min_diss`. It consists of the following pseudo-code:

Algorithm 1 Find shift warping minimizing dissimilarities

1: Given a list of curves $\{\mathbf{x}_i\}_{i=1}^N$, a list of cluster centers $\{\mathbf{v}_k\}_{k=1}^K$ and a vector of cluster center lengths $[c_k]_{k=1}^K$:
2: **for** k = 1,...,K **do**
3: **for** i=1,...,N **do**
4: Find the shift minimizing the dissimilarity between the shifted $\mathbf{x}_i$ and $\mathbf{v}_k$:
5: For each eligible shifts $s_{k,i}$ construct the matrix $\tilde{\mathbf{x}}_{i,s_{k,i}}$;
6: **if** There exists an matrix with a number of nan-free rows $\geq c_k$ **then**
7: Find among these the shift $\hat{s}_{k,i}$ that minimizes $d_\alpha^2(\tilde{\mathbf{x}}_{i,s_{k,i}}, \mathbf{v}_k)$;
8: **else**
9: Take as minimizing shift the one corresponding to the matrix with the maximum number of nan-free rows;
10: **end if**
11: Update $S_{i,k} = \hat{s}_{k,i}$;
12: **end for**
13: **end for**

In the previous `R` implementation, this part of the code is parallelized using the *R parallel*. This approach consists of generating all possible combinations between the list of curves and the list of motifs and then parallelizing the iterations on those combinations in which points $4-10$ of the algorithm are executed.

Since we cannot parallelize the two external for loops in `C++`, we decided to distinguish the two cases: if the algorithm is executed sequentially, avoid generating the big list of combinations between motifs and curves and run the `C++` implementation of the above algorithm, if, instead the code is executed in parallel rely on the `R` parallelism executing the `C++` implementation of points $4-10$ of the algorithm.

Another part of the probKMA function we implemented is the *center elongation* section. Here, the domain of the motifs is elongated according to the following procedure:

Algorithm 2 Center elongation

for k = 1,...,K **do**

Find the maximum elongation length `len_max_elong_k` as

$$\min(\lfloor \text{length}(\mathbf{v}_k)\ \texttt{max\_elong} \rfloor, \texttt{c\_max} - \text{length}(\mathbf{v}_k))$$

where `c_max` and `max_elong` are user-input parameters;

Since the number of maximum attempts to elongate each motif is specified by the user through the parameter `trials_elong`, extend the $\mathbf{v}_k$ domain of a number of rows equal to:

if `len_max_elong_k` $\leq$ `trials_elong` **then**

$$1 : \texttt{len\_max\_elong\_k}$$

else

$$\text{round}(\text{linspace}(1 : \texttt{len\_max\_elong\_k}, \texttt{trials\_elong}))$$

end if

Modify the shifts in $\mathbf{s}_k := \{s_{k,i}\}_{i=1}^N$ according to the elongated domains;

For all the elongated domains compute the new motifs using the `R` function `compute_motif` , obtaining a list of possible elongated motifs;

Using the `C++` function `compute_Jk_rcpp`, compute the functional related to the original cluster center `Jk_before` and to all the elongated ones `Jk_after` as following

$$\sum_{i=1}^{N} (p_{k,i})^m \ d_\alpha^2(\tilde{\mathbf{x}}_{i,s_{k,i}}, \mathbf{v}_k)$$

if $\min(\texttt{Jk\_after} - \texttt{Jk\_before}/\texttt{Jk\_before}) < \Delta_{Jk}$ **then**

Return the elongated motif that achieves the smallest percentage change in the functional;

end if

end for

Its implementation is contained in the `C++` functions `elongation_rcpp` and `elongate_motifs` in the file *elongate_ motifs.cpp*. They are both *void* functions since they take as input the list of motifs, their domains, and the shifts as *non-const reference* and modify directly them with the elongated version if the conditions are fulfilled.

In addition, we implemented the following functions external to the probKMA algorithm:

- `probKMA_silhouette_rcpp` which computes the silhouette indexes as briefly explained in Section 2.7. The function core consists of extracting from all the curves the sub-matrices (pieces of the curve) assigned to the clusters they belong using the dichotomized membership probability matrix `P_clean`. Next, the distance between each pair of extracted pieces is computed. If the `align` flag is true, the distance between the two pieces is calculated by finding the best shift to apply to the longest curve that minimizes the distance from the second extracted segment. If `align` is false and the two curves have the same length then shifting is not applied.

  The previous `R` implementation used the `combn` function, which takes as input a list of curve pieces and returns all possible combinations of two elements. In our implementation `C++`, we avoided creating such a large data structure by combining the indexes corresponding to their position in the list containing them. To do so, we implemented the following `C++` function, which can be found in the *utilities.hpp* header file:

```
template<typename T>
arma::Mat<T> combn2(const arma::Col<T> & y){
int n = y.n_elem;
arma::uvec v(n,arma::fill::zeros);
v(0) = 1;
v(1) = 1;
arma::uword l = 0;
arma::Mat<T> result(2,n*(n-1)/2, arma::fill::zeros);
arma::uword k;
do {
  k = 0;
  auto filter_index = std::views::iota(0,n)
    | std::views::filter([&v](int i){return v(i);});
  for(auto i: filter_index)
    result(k++,l) = y(i);
  l++;
} while (std::prev_permutation(v.begin(), v.end()));
return result;
}
```

This function allows us to obtain all the combinations relying on the function of the standard library `std::prev_permutation`. In fact, every combination of two elements of the list of indexes `y` can be seen as the permutation of a mask with the same size of `y` and containing all `false` values except for two `true` positions;

- `motif_search` which identifies and ranks motifs within functional datasets. This involves evaluating the similarity between each candidate motif and segments of the functional data. Functional curves are grouped into clusters using a global radius parameter `R_all`.

  The clustering is determined by cutting the dendrogram at a height equivalent to twice the global radius. Then, if provided, a vector of group-specific radii `R_m` is used for each cluster. Alternatively, if `R_m` is set to `NULL`, the function determines group-specific radii based on distances between motifs within the same group and all functional curves. Within each group, motifs are ranked based on their frequencies and radii. The function prioritizes motifs with higher occurrence frequencies and smaller radii. In `C++` this was achieved by defining two utilities which can be found again in the *utilities.hpp*:

  - `avg_rank` computes the average rank of elements, implemented as follows:

```
template<typename T,typename Comparator>
arma::vec avg_rank(const T& x)
{
  R_xlen_t sz = x.size();
  Rcpp::IntegerVector w = Rcpp::seq(0, sz - 1);
  std::sort(w.begin(), w.end(), Comparator(x));

  arma::vec r;
  r.set_size(sz);
  for (R_xlen_t n, i = 0; i < sz; i += n) {
    n = 1;
    while (i + n < sz && x[w[i]] == x[w[i + n]]) ++n;
    for (R_xlen_t k = 0; k < n; k++) {
      r[w[i + k]] = i + (n + 1) / 2.;
    }
  }
  return r;
}
```

- `order2` which returns the order or rank of the elements and has the following implementation:

```
template<typename V,typename T>
T order2(const V& x, bool desc = false)
{
    std::size_t n = x.size();
    T idx;
    if constexpr(std::is_same<V, arma::uvec>::value)
    {
      idx.set_size(n);
    }
    else
    {
      idx = T(n);
    }
    std::iota(idx.begin(), idx.end(), static_cast<size_t>(1));
    if (desc) {
      auto comparator =
      [&x](size_t a, size_t b){ return x[a - 1] > x[b - 1]; };
      std::stable_sort(idx.begin(), idx.end(), comparator);
    } else {
      auto comparator =
      [&x](size_t a, size_t b){ return x[a - 1] < x[b - 1]; };
      std::stable_sort(idx.begin(), idx.end(), comparator);

      size_t nas = 0;
      for (size_t i = 0; i < n; ++i, ++nas)
        if (!Rcpp::Vector<REALSXP>::is_na(x[idx[i] - 1])) break;
      std::rotate(idx.begin(), idx.begin() + nas, idx.end());
    }
    return idx;
}
```

The selection of motifs within each group is determined by choosing the motif with the highest frequency and the lowest mean dissimilarity. This dual criterion ensures that selected motifs are both prevalent and representative of the functional patterns in the group. To enhance diversity, additional motifs may be selected within a group if their functional forms differ significantly from the initially chosen motif. This minimum difference in functional form is specified by the `length_diff` parameter. Lastly, the dissimilarity between a candidate motif and a functional curve is calculated, and motifs are considered to match a curve if the dissimilarity falls below the corresponding group-specific radius `R_m`.

4.3. Public interface

The user interface of this first package is equivalent to that of the previous implementation `R`. Here, we list and briefly explain what the various functions visible in the package do. The documentation, generated with *Roxygen2*, also contains a description of the input and output of each function.

- `probKMA`: probabilistic k-means with local alignment to find candidate motifs;
- `probKMA_plot`: plot the results of `probKMA`;
- `find_candidate_motifs`: run multiple times `probKMA` function with different numbers of motifs, minimum motifs length and initialization, to find a set of candidate motifs;
- `filter_candidate_motifs`: filter the candidate motifs based on a threshold on the average silhouette index and a threshold on the size of the curves in the motif;
- `cluster_candidate_motifs`: determine a global radius, group candidate motifs based on their distance, and determine a group-specific radius;
- `cluster_candidate_motifs_plot`: plot the results of `cluster_candidate_motifs`;
- `motifs_search`: find occurrences of the candidate motifs in the curves and sort them according to their frequencies and radius;
- `motifs_search_plot`: plot the results of `motifs_search`.

5. Second Implementation

The second package we implemented contains the function `probKMA` entirely rewritten in `C++`. We decided to rewrite such because it is the part that impacts the most in the computational time of the library and because, especially for functional data that are big matrix and with a large number of rows, a speed-up in `probKMA` allows the execution of `find_candidate_motifs` in reasonable time. Rewriting `probKMA` allowed us more flexibility in choosing data structures and code design. Furthermore, by minimising the use of structures from the *Rcpp* library, it was possible to parallelise the code using *openMP*.

5.1. Data Structures

The *Armadillo* library provides the `arma::field` data structure. The `arma::field` allows you to store elements of different types: matrices, column vectors and row vectors. In our case, the `_ProbKMAImp` class, which we will explain in detail in Section 5.3, has as a data member an object `_Y` of class `arma::field`. In particular, `_Y`(i,j) contains a matrix of the class `arma::mat` that corresponds to the curve $\mathbf{x}_i$ or its derivative $\mathbf{x}'_i$. Since the matrices in different field rows might have different numbers of rows, a field provides flexibility in handling these structures within a single container. In the initialisation of the `_ProbKMAImp` class, the user determines what type of distance she wants to consider: if $\alpha \in (0, 1)$, the constructor of `_ProbKMAImp` allocates a field with two columns to hold both the derivatives and the curves, whereas if $\alpha = 1$ only the derivatives will be stored in `_Y`, which will only have one column, as well as in the case $\alpha = 0$, where `_Y` will only contain the curves. We adopted the same approach to handle cluster centers in the `_V` data member of `_ProbKMAImp`.

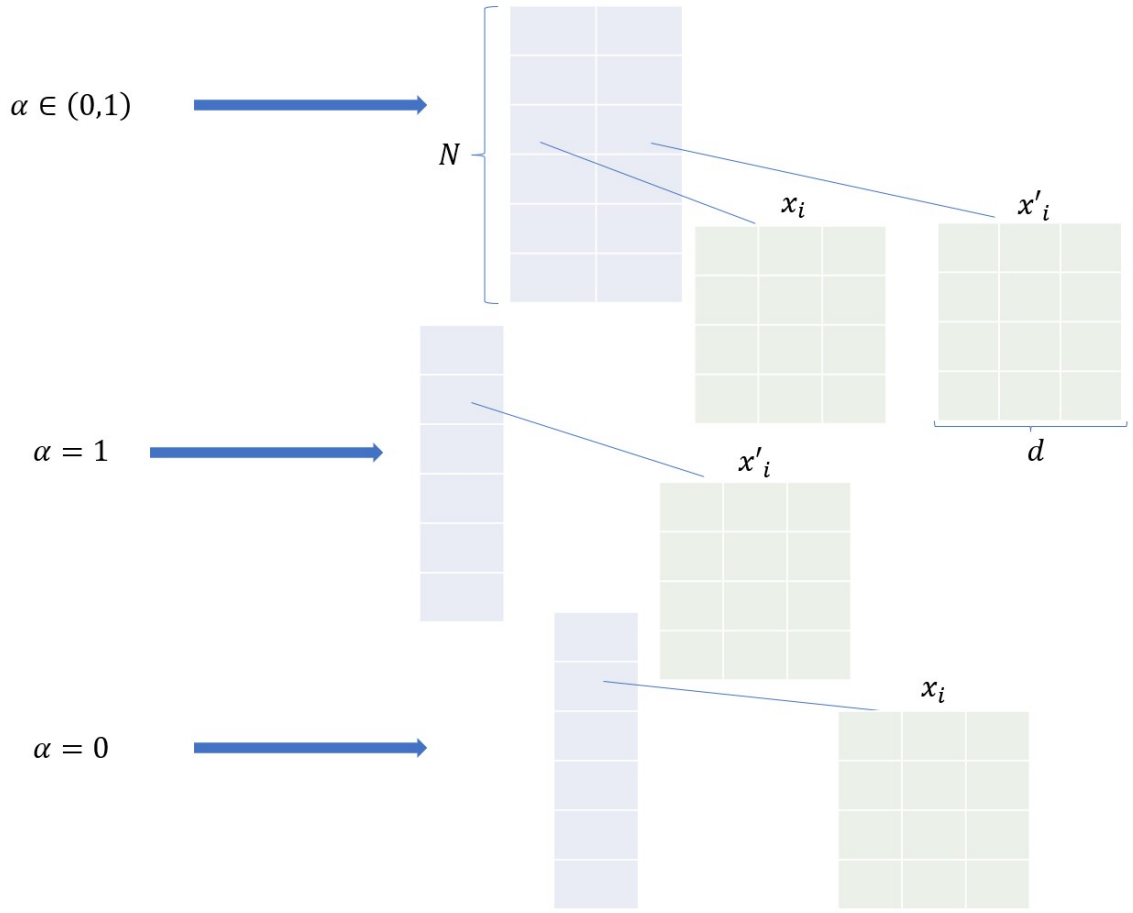

Figure 2: Field data structure.

5.2. Algorithm workflow

As illustrated in the Figure 3, the algorithm is structured as follows:

1. The user provides as input the parameters needed to execute ProbKMA to the function `initialChecks`, which takes care of checking the correctness of the inputs and the data, modifying some of them if necessary, e.g. in the case of null or invalid matrices `P0` and `S0` these are generated randomly. In particular, `InitialChecks` returns a list of two elements: the `Parameters` list and the `FuncData` list, containing the parameters and a list with the functional data and the initial matrices `P0` and `S0` respectively;
2. Since the `C++` class `ProbKMA` is exported in `R`, the "problem loading" step consists of instantiating an object of this class in `R`. Following the *PImpl* design pattern, the `ProbKMA` class definition includes a `std::unique_ptr` pointer to a forward-declared class `_probKMAImp`. This class contains all the private implementation details and, its constructor is called when the public class `ProbKMA` is constructed. The constructor of the class `_probKMAImp` handles the casting of `R` data structures into `C++` data structures. It also creates the various *factories* that instantiate the `std::shared_ptr` pointers to the objects of the classes `MotifPure`, `Dissimilarity` and `PerformanceIndexAB`, which will be explained in Section 5.3;
3. Once we have an istance of the `probKMA` class, the algorithm can be run through the `probKMA_run` function. The function `probKMA_run` relies on the actual implementation in the class `_ProbKMAImp`, contained within the method of the same name. The run consists of the following main parts executed iteratively:
   (a) *compute motifs*: clusters centers are updated as explained in Section 2.2. This part is managed by the `MotifPure` class;
   (b) If the algorithm is close to convergence, the *elongation center* part is performed. The `MotifPure` class handles this part;
   (c) *find shift warping minimizing dissimilarities*: for each $k = 1, ..., K$ and each $i = 1, ..., N$ the `Dissimilarity` class handles the issue of finding the best shift minimizing the distance between motif k and curve i;
   (d) *compute membership*: the membership probability matrix is updated, as explained in Section 2.2;
   (e) *compute Bhattacharya distance*: The GDB, used for the stop criterion, is computed.
4. The run ends with a *prepare output* section, which performs a cleaning step, as explained in Section 2.5 and returns a list containing all the outputs of the algorithm and some of the input parameters.

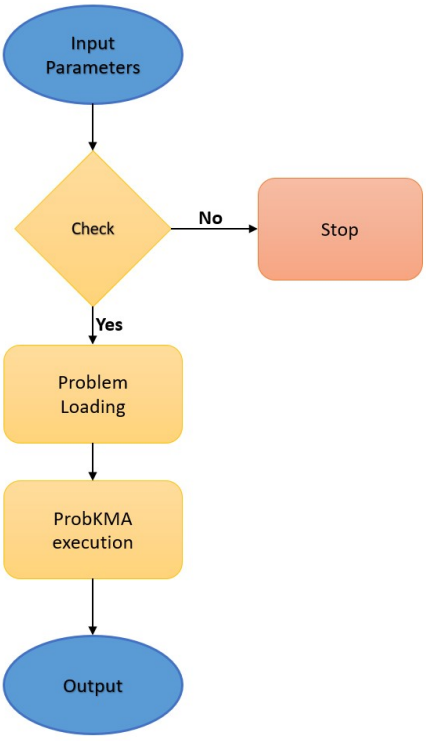

Figure 3: Algorithm's workflow.

If the user wants to execute the probKMA algorithm with new parameters, it is not necessary to instantiate a new object of class `ProbKMA`. It is sufficient to use the method `set_parameters`, which changes the parameters of `_probKMAImp` and its data members.

5.3. Classes

In this section, we briefly explain the classes we implemented.

- `Dissimilarity`: an abstract class that forms the basis for different types of functional data distance. Its derived class `SobolDiss` is specific to distances of type Sobolev, considered up to now in the ProbKMA algorithm. Its method `distance`, which takes in input two curve segments of the same length, deals with the computation of the L^2 square metric. The template method `find_diss_helper` finds the shift that minimises the distance between each curve and the considered cluster center. The template method `computeDissimilarityClean_helper` calculates the `D_clean` matrix requested by the *prepare output* part of probKMA. The methods of `SobolDiss` are then employed by its child classes: `L2` and `H1`, used in the cases of $\alpha = 0$, $\alpha = 1$ and $\alpha \in (0, 1)$ respectively;

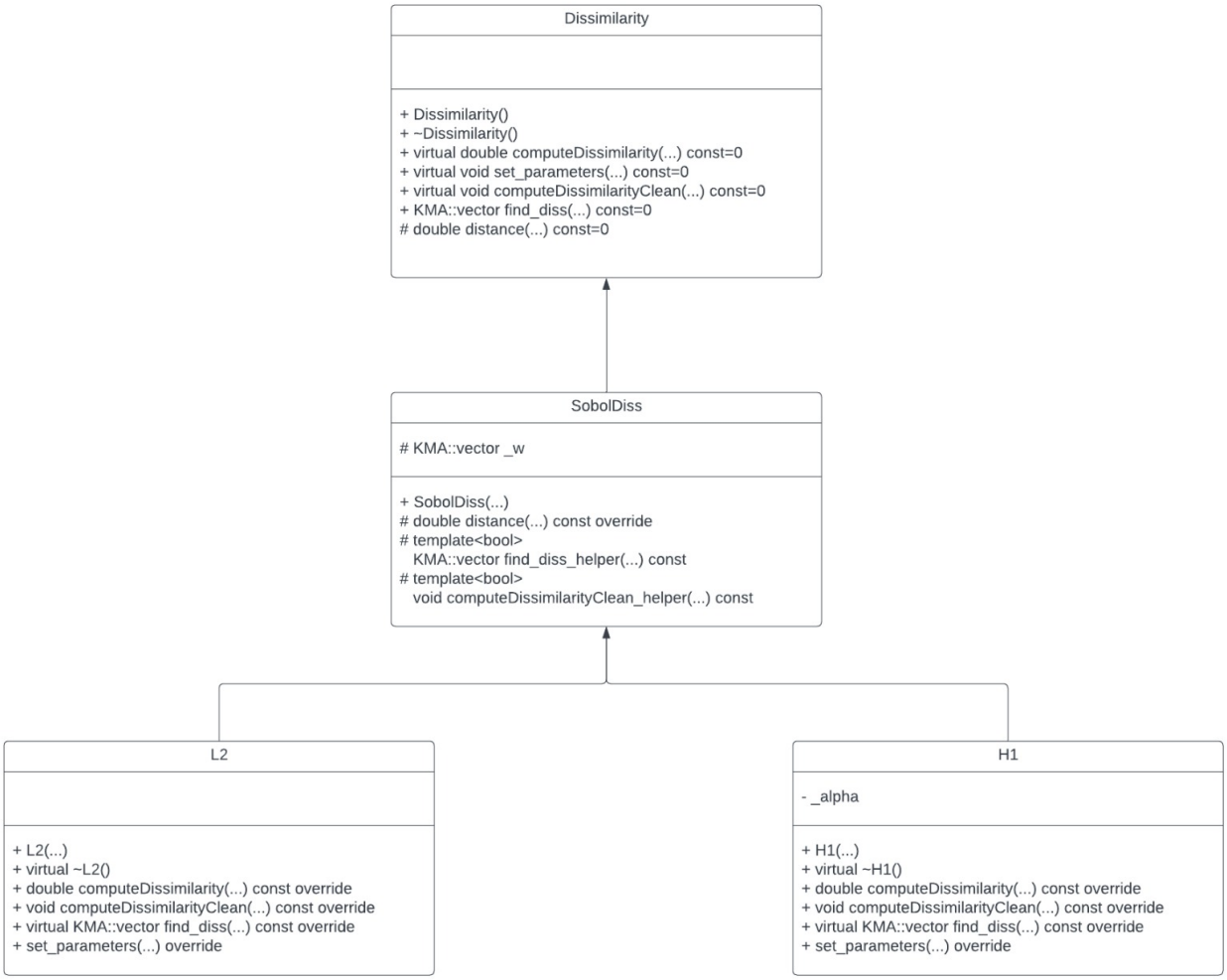

Figure 4: Inheritance diagram for `Dissimilarity`.

- `MotifPure`: an abstract class that forms the basis for updating and elongating the motifs according to different types of distance. Its derived class `MotifSobol` is specific to distances of type Sobolev, considered up to now in the ProbKMA algorithm. Its method `compute_motif_helper` compute the new motif k based on $\mathbf{s}_k$ and p_k while `elongate_motifs_helper` performs the *elongation center* step. The methods of `MotifSobol` are then employed by its child classes: `MotifL2` and `MotifH1`, used in the cases of $\alpha = 0$, $\alpha = 1$ and $\alpha \in (0, 1)$ respectively;

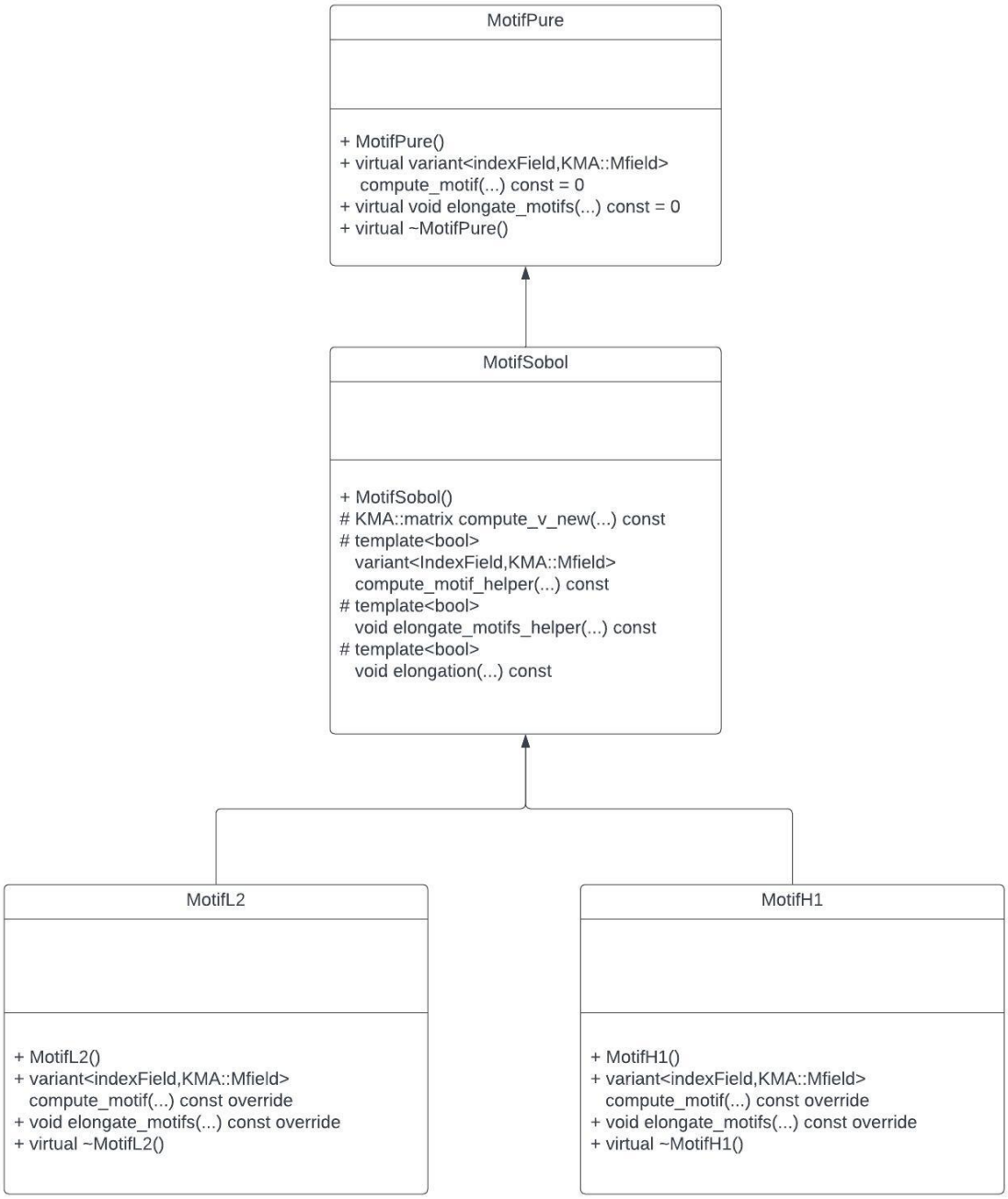

Figure 5: Inheritance diagram for `MotifPure`.

- `PerformanceIndexAB`: an abstract class that forms the basis for different types of performance index, i.e. the functional (2). It contains the template method `shiftCurveHandle`, which given the curves and the shifts related to motif k find $\tilde{\mathbf{x}}_{i,s_{k,i}}$ for all $i = 1, ..., N$. Its derived class `MotifSobol` is specific to distances of type Sobolev and contains the method `compute_Jk_helper`, which computes the following:

$$\sum_{i=1}^{N} (p_{k,i})^m \; d_\alpha^2(\tilde{\mathbf{x}}_{i,s_{k,i}}, \mathbf{v}_k)$$

  The methods of `MotifSobol` are then employed by its child classes: `PerformanceL2` and `PerformanceH1`, used in the cases of $\alpha = 0$, $\alpha = 1$ and $\alpha \in (0, 1)$ respectively;

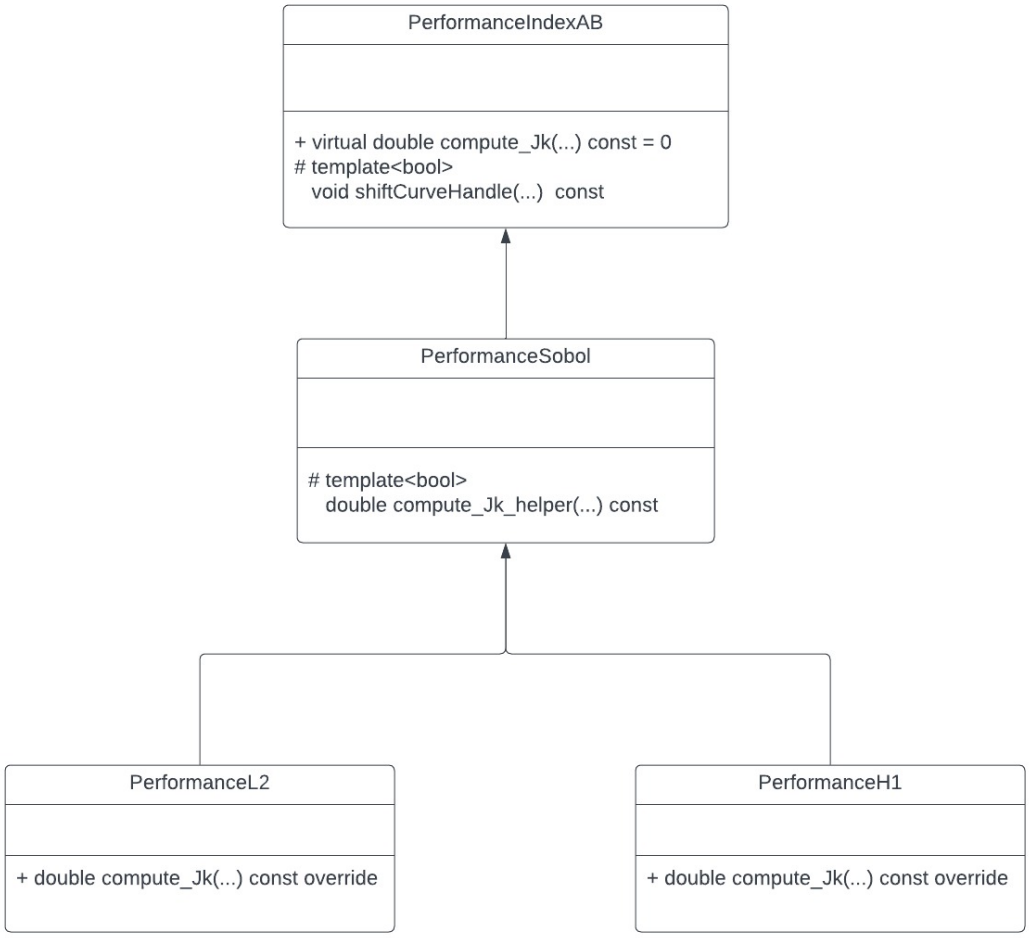

Figure 6: Inheritance diagram for `PerformaceIndexAB`.

- `ProbKMA`: a class that executes the probKMA algorithm. According to the *PImp* design pattern, it contains a pointer to the forward-declared class `_probKMAImp`. The PImpl idiom allows us to achieve better encapsulation, reduce compile-time dependencies, and provide a more stable interface. All the implementation details are contained in `_probKMAImp`, while `ProbKMA` is the user interface, used to run the algorithm through the `probKMA_run`, setting new parameters with the `set_parameters` function and set the initial matrices `P0` and `S0`;

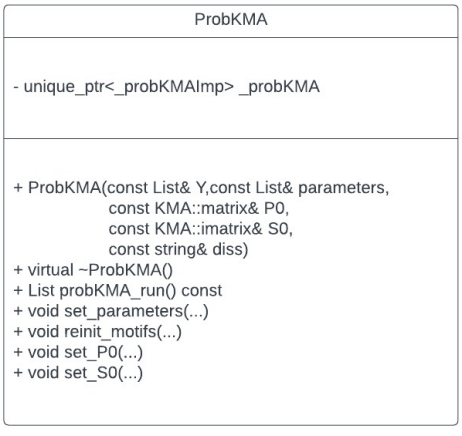

Figure 7: Class diagram for `ProbKMA`.

- `_probKMAImp`: a class that handles all the implementation details of the probKMA algorithm. It has the following data members:
  - `_Y`: field containing the functional data;
  - `_V`: field containing the cluster centers, updated at each iteration of the algorithm;
  - `_motfac`: a pointer to the `MotifPure` class;
  - `_dissfac`: a pointer to the `Dissimilarity` class;
  - `_perfac`: a pointer to the `PerformanceIndexAB` class;
  - `_P0`: initial membership probability matrix;
  - `_S0`: initial shift probability matrix;
  - `_isY0`: a boolean that is `true` if and only if the functional data contains the curves;
  - `_isY1`: a boolean that is `true` if and only if the functional data contains the derivatives;

  The method `probKMA_run` implements the probKMA algorithm and returns a big `Rcpp::list` to `R` through the `to_R` function. The output list comprises some of the input parameters and the following elements:
  - `V0`: a list with the motifs found;
  - `V1`: a list with the derivative of the motifs found;
  - `V0_clean`: a list with the motifs found after an additional cleaning step;
  - `V0_clean`: a list with the derivatives motifs found after an additional cleaning step;
  - `P_clean`: the cleaned membership matrix;
  - `S_clean`: the cleaned shifts matrix;
  - `D`: the dissimilarity matrix, i.e. $D(i,k) = d_\alpha^2(\tilde{\mathbf{x}}_{i,s_{k,i}}, \mathbf{v}_k)$;
  - `J_iter`: a vector with the functional computed at each iteration;
  - `BC_dist_iter`: a vector with the GDB at each iteration;
  - `iter`: number of iterations performed;
  - `P`: computed membership matrix;
  - `S`: computed shift warping matrix.

  The `Initialize` function is used in the class constructor to cast functional data from `Rcpp::List` to `arma::field` and uses the methods: `handleCaseH1`, `handleCaseL2` and `handleNonNullY`. The `reinit_motifs` method initializes the motifs to an empty field with the requested dimension. In addition, it is used by the user before running (after the first time) the algorithm to set the new dimension of the motifs.

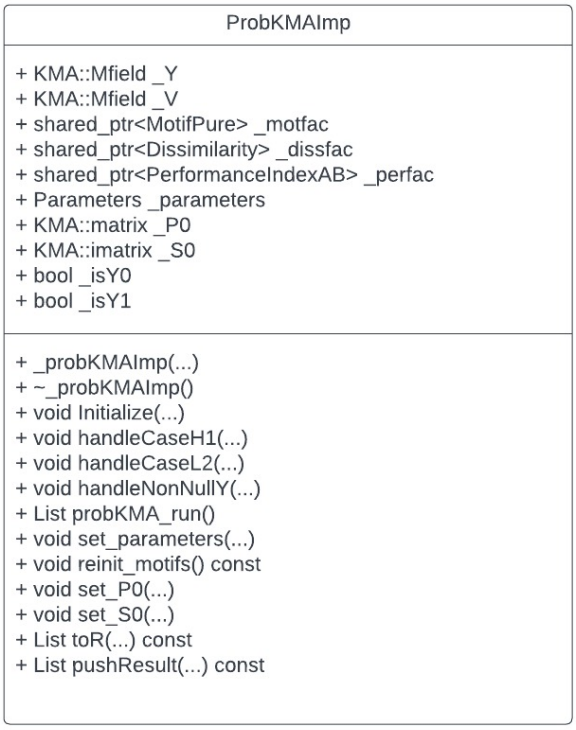

Figure 8: Class diagram for `_probKMAImp`.

- `factory`: it provides a mechanism for creating and managing instances of classes derived from a common base class. It allows dynamic registration of derived classes with associated factory functions, and later, it can instantiate these classes based on string identifiers. In particular, the following are defined:
  - `instantiate`: it instantiates an object of a derived class based on the provided name. It looks up the name in the map and calls the associated factory function to create and return a shared pointer to the derived class instance;
  - `FactoryRegister`: a template function to register a derived class `D` in the factory. It associates the provided name with a lambda function that constructs an instance of `D` using the arguments and stores it in the `map`.

```
template<typename B>
class SharedFactory
{
public :

  using registry_map =
        std::unordered_map<std::string_view,
                           std::function<std::shared_ptr<B>()>>;

  registry_map map;

  std::shared_ptr<B> instantiate(std::string_view name)
    {
      auto it = map.find(name);
      return it == map.end () ? nullptr : (it->second)();
      }

 template <typename D, typename... Args>
 void FactoryRegister(std::string_view name, Args&&... args) {
   map[name] =
   [args = std::forward_as_tuple(std::forward<Args>(args)...)]()
            mutable
     {
       return std::apply([](auto&&... a)
      {return std::make_shared<D>(std::forward<decltype(a)>(a)...);},
                                    std::move(args));
     };

 }
};
```

Specifically, the following code is used to register all base classes:

```
// Create Dissimilarity factory
util::SharedFactory<Dissimilarity> dissfac;
dissfac.FactoryRegister<L2>("L2",_parameters._w);
dissfac.FactoryRegister<H1>("H1",_parameters._w,_parameters._alpha);

// Create Motif factory
util::SharedFactory<MotifPure> motfac;
motfac.FactoryRegister<MotifL2>("L2");
motfac.FactoryRegister<MotifH1>("H1");

//Create Performance factory
util::SharedFactory<PerformanceIndexAB> perfac;
perfac.FactoryRegister<PerformanceL2>("L2");
perfac.FactoryRegister<PerformanceH1>("H1");

//check whether diss is valid
_motfac = motfac.instantiate(diss);
_dissfac = dissfac.instantiate(diss);
_perfac = perfac.instantiate(diss);
```

This class structure makes the code versatile and easily extendable. Introducing new distance or functional types does not mandate alterations to the existing implementation of the algorithm. Instead, the process involves implementing new classes dedicated to handling these extensions. These new classes will inherit from the corresponding abstract classes and subsequently be registered by the factories within the `_probKMAImp` constructor.

5.4. Public Interface

To run the probKMA algorithm, the user gives as input to the function `InitialChecks` the following data:

- `Y0`: list containing the curves;
- `Y1`: list containing the derivatives (if $\alpha \neq 0$);
- `P0`: initial membership probability matrix;
- `S0`: initial shift warping matrix;
- `diss`: a string indicating the type of distance;
- `params`: a list with the additional parameters (see the documentation for an explanation of each element it contains);
- `seed`: the seed used for reproducibility to initialize randomly `P0` and `S0`, if these matrices are `NULL`.

If the inputs are admissible, InitialChecks will return a list containing two lists: `Parameters` and `FuncData`. At this point, the user creates a pointer to the `ProbKMA` class with the following command:

```
prok = new(ProbKMA,data$Y,params,data$P0,data$S0,metric)
```

if $\alpha = 0$ or $\alpha = 1$ then `metric = "L2"`, otherwise `metric = "H1"`.

The pointer to the `prok` object can be used to execute the probKMA algorithm in the following way:

```
output <- prok$probKMA_run()
```

At this point, if the user wants to modify the parameters, she may set them directly using the pointer `prok` as follows:

```
prok$set_parameters(params)
```

Then, she has to re-initialize the motifs with the right dimensions:

```
prok$reinit_motifs(params$c,ncol(Y0[[1]]))
```

and run the algorithm as above.

5.5. Parallelism

The multi-thread algorithm's version utilizes *OpenMP*. To ensure compatibility with compilers lacking *OpenMP* support, each omp directive is safeguarded by the following guarding header, as in the case of the header inclusion:

```
# ifdef _OPENMP
# include <omp.h>
# endif
```

The steps of the algorithm that can be executed in parallel are the *find shift warping minimising* part and the *elongation centre* part, as they are the most computationally expensive parts.

Concerning *find shift warping minimising* section, we parallelized as follows:

```
#ifdef _OPENMP
   #pragma omp parallel for collapse(2) firstprivate(sd)
                                        num_threads(n_threads)
#endif
for (arma::uword i = 0; i < _n_rows_V; ++i)
   for (arma::uword j = 0; j < _n_rows_Y; ++j)
   {
       sd = _dissfac->find_diss(_Y.row(j),_V.row(i),w,alpha,c_k(i));
       S(j,i) = sd(0);
       _D(j,i) = sd(1);
   }
```

The `collapse(2)` clause is applied to the `#pragma omp parallel for` directive. It indicates that both loops i and j should be collapsed into a single loop for parallelization. Since generally the number of motifs to search is at most 5, i.e. `_n_rows_V = 5` while the number of threads is greater, without `collapse(2)` some threads would not be used. But when the loops are collapsed, the threads will be distributed between `_n_rows_V * _n_rows_Y` iterations which is certainly much greater than the number of threads (`_n_rows_Y` is the number of curves in the dataset).

As for the elongation step, we simply parallelized the iteration over the number of motifs, since finding the best possible elongation requires a large number of operations.

```
#ifdef _OPENMP
     #pragma omp parallel for num_threads(n_threads)
 #endif
 for (unsigned int i = 0; i < V_dom_size; ++i){
   elongation<use1>(V_new,V_dom,S_k,P_k.col(i),
                    len_elong[i],keep.col(i),param._c[i],
                    Y,i,param,perf,diss);
 }
```

5.6. Documentation

The documentation is created using *Roxygen2* for `R` packages and is available in the **man directory**. For the `C++` code, additional documentation is created by `Doxygen` and available at: `https://niccolof.github.io/ProbKMA-FMD/`, where we describe the various classes implemented and explain the various methods, their inputs and outputs.

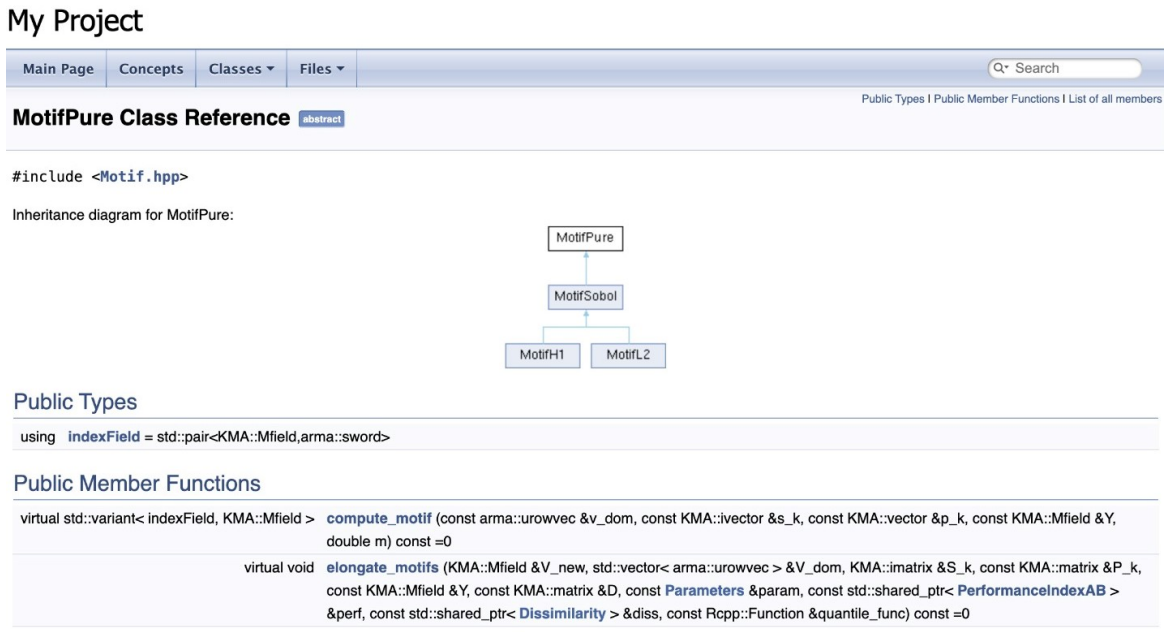

Figure 9: Documentation website.

6. Simulations

In this section, we show some applications of the two packages to examples of functional datasets. The datasets are available within the GitHub repository. We compare the computational times required by the previous `R` implementation and our two implementations for scalar functional data and vector value data.

The simulated data are contained in the **Data directory** of the two packages, under the names: `sim_motifs` for the first package implementation and `simulated200` for the second package implementation. Loading data contained in the **Data directory** is possible using the following `R` command:

`load(...)`

where `...` has to be replaced by: `sim_motifs` or `simulated200` depending on the package. The High-Dimensional Data are instead contained in **Test_probKMA** directory in branch `main_1` under the name `Y.RData` and in **Test_comparisons** directory in branch `main_3` under the same name `Y.RData`.

6.1. Simulated Data

The work by Cremona and Chiaromonte [1] explains how to generate curves comprising functional motifs. The approach employs a B-spline basis with equally spaced knots and defines each curve as a combination of basis functions with associated coefficients. The order of the B-spline basis controls the smoothness and complexity of the curves. The motifs are created by fixing the values of subsequent coefficients of this expansion. Motif occurrences that are the same, both in shape and level, are generated by adding Gaussian noise $\mathcal{N}(0, \sigma^2)$ to the corresponding coefficients. Motif occurrences that are the same in shape but have different levels are obtained by adding a constant to all the coefficients that define a single occurrence, choosing different constants for different occurrences.

The simulated data we have used consists of 20 curves of length 200 (corresponding to 201 evaluation points) with 2 different motifs of length 60 (corresponding to 61 evaluation points).

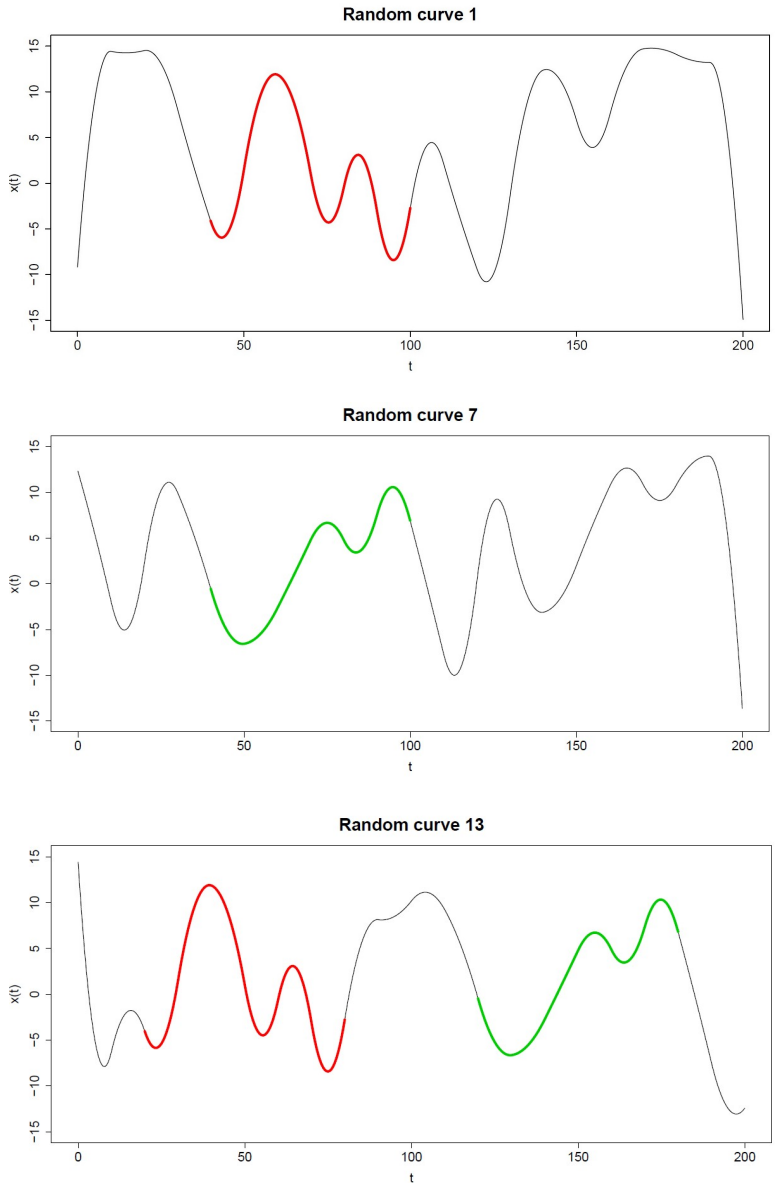

Figure 10: Plot of some curves in the dataset: the red segments denotes the occurrence of motif 1, while the green segments denotes the occurrence of motif 2.

Concerning the presence of the motifs in the curves, we have:

- Curves $1-6$ contain one occurrence of motif 1;
- Curves $7-12$ contain one occurrence of motif 2;
- Curves $13-14$ contain one occurrence of motif 1 and one occurrence of motif 2;
- Curves $15-16$ contain two occurrences of motif 1;
- Curves $17-18$ contain two occurrences of motif 2;
- Curves $19-20$ contain no motif.

The data have been generated using B-spline basis of order 3 and knots at distance 10. The standard deviation of the gaussian noise added to the motif occurrences is $\sigma = 0.1$.

The scripts used to perform the comparisons are:

- branch: `main_1`, folder: **Test_comparisons**, script: `Comparisons_vectorial_data.R`;
- branch: `main_3`, folder: **Test_probKMA**, script: `Comparison_probKMA_vector.R`.

In these tests, the times of a single run of the probKMA algorithm and the function `find_candidate_motifs` in which probKMA is run many times with different initialisations are compared. We emphasise that for the results of `find_candidate_motifs` to be comparable with the previous implementation `R`, it was necessary to modify this function by setting, within the loop in which `probKMA` is called, the seed, used to generate the matrices `P0` and `S0`. In the second implementation, the function `find_candidate_motifs` has only been adapted to the new implementation `C++` and not rewritten. We suppose rewriting this function in `C++` could lead to a higher speed-up.

Furthermore, we notice that in the full `C++` implementation of probKMA, there are errors on the order of at most 10^{-15} due to the different levels of accuracy of some functions used for the distances computations such as `sum` of `R` and the correspondent `arma::accu` of *Rcpp Armadillo*.

The results obtained are the same as the previous `R` implementation. Figure 11 shows the plots of the cluster center computed with a run of the probKMA algorithm. We can see that the shape of the motifs has been correctly identified and the shifted curve segments align perfectly with the clusters to which are assigned.

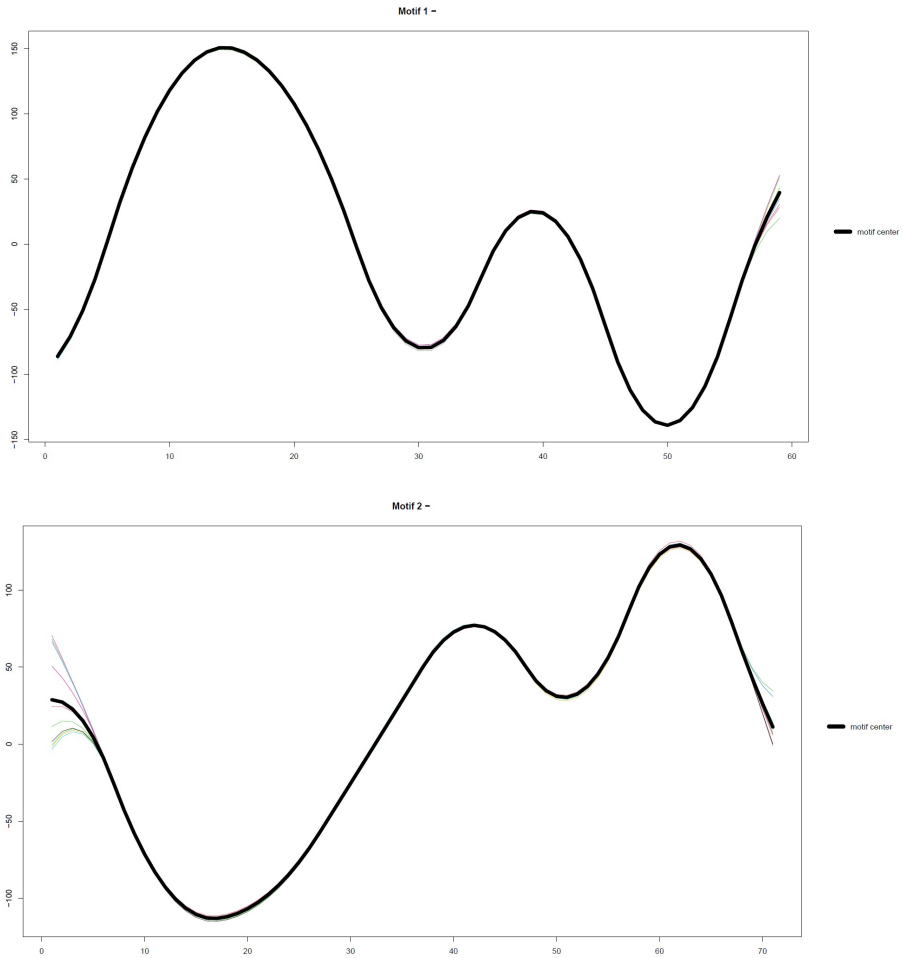

Figure 11: Cleaned motif detected and segments of the curves aligned according to the shift matrix.

Figure 12 represents two barplots showing for each motif the membership probabilities obtained for the curves in the dataset. The curves are assigned mostly correctly. The probability of belonging to one of the two clusters for curves 19 and 20 with no motif occurrences is not particularly unbalanced in favour of either motif. The curves with occurrences of both motifs are assigned with a higher probability to the first cluster but still have a positive probability of belonging to the second one. Better results can be obtained by following the post-processing steps, as explained in Section 2.8.

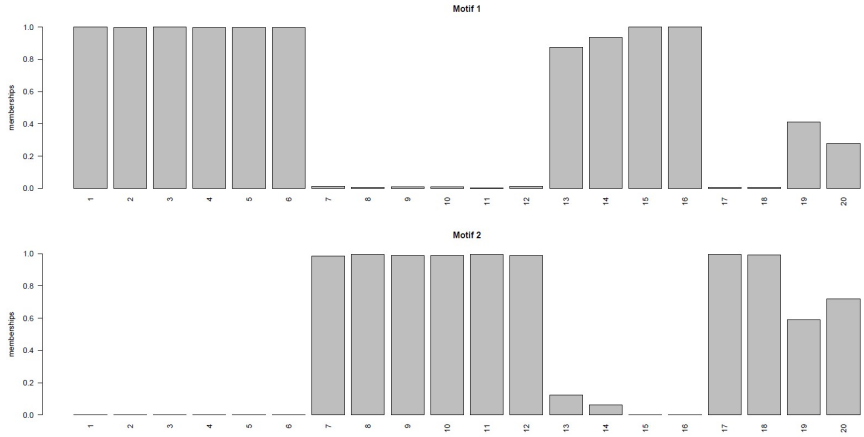

Figure 12: Barplots for the membership probabilities.

Table 1 represents the comparison of computation times of the probKMA algorithm between the three different implementations. Times are expressed in seconds: *user* indicates the CPU time used by the user, *system* the CPU time used by the operating system and *elapsed* the total time from start to finish. For the `C++` package, we have also included the times varying the number of threads used in the parallel parts of the code.

Table 1

	user	system	elapsed
`C++` (`n_threads = 7`)	0.471	0.001	0.101
`C++` (`n_threads = 3`)	0.222	0.002	0.097
`C++` (`n_threads = 1`)	0.167	0.009	0.150
`R/C++`	2.447	0.183	7.99
`R`	6.980	2.495	33.397

Table 1: Speed Comparison probKMA algorithm: `C++` refers to the second package, `R/C++` to the first package and `R` to the previous `R` implementation.

Table 2 represents the comparison of computation times of the `find_candidate_motifs` functions between the three different implementations.

Table 2

	user	system	elapsed
`C++`	0.250	0.152	53.731
`R/C++`	0.200	0.161	574.302
`R`	6.362	1.814	1331.180

Table 2: Speed Comparison `find_candidate_motifs`: `C++` refers to the second package, `R/C++` to the first package and `R` to the previous `R` implementation.

Let us note that in the case of the second package, parallelism using *openMP* involves only `probKMA_run`, and we have parallelized the loop in which probKMA is executed several times in `R` using *R parallel*. In both cases, the number of threads is equal to 7. In the other two implementations, parallelism in `R` is present with 7 threads both in `probKMA` and in the loop of `find_candidate_motifs`.

6.2. High-Dimensional Data

The data used in this section are the bivariate version of the mutagenesis data used in the paper [1]. In particular, we obtain them by combining *substitution* rates with *indel* rates. *Substitution* rates refer to the frequency at which one nucleotide is replaced by another in a DNA sequence. *Indel* (insertion and deletion) rates are the frequencies of insertion or deletion of nucleotides in a DNA sequence. The discovery of functional motifs within these data is of great interest in genomic science because it provides insights into mutagenesis phenomena, which are involved in many diseases such as the development of cancers.

The functional dataset consists of 43 matrices for curves and an equal number for their derivatives of two columns and a variable number of rows ranging from a minimum of 1836 to a maximum of 43868. The data presents a lot of missing values (large gaps).

Regardless of the interpretation that the motifs found may have, for which we refer to the work of Cremona and Chiaromonte [1], these data allowed us to test the code in its generality due to the presence of gaps and curves of different lengths. Moreover, our code application to this dataset makes us realize the impact that the size of the functional data has on the execution of the probKMA algorithm.

The scripts used to perform the comparisons are:

- branch: `main_1`, folder: **Test_comparisons**, script: `Comparisons_matrices_data.R`;
- branch: `main_3`, folder: **Test_probKMA**, script: `Comparison_probKMA_matrix.R`.

The parameters chosen require more than 200 iterations for the algorithm to converge, so three simulations were run by interrupting the execution of probKMA at the following iterations: 20, 100 and 200. The Tables 3,4,5 represent the computational times for `iter_max` = 20, `iter_max` = 100 and `iter_max` = 200 respectively.

Table 3

	user	system	elapsed
C++ (n_threads = 7)	32.00	0.37	5.70
C++ (n_threads = 3)	21.49	0.07	7.69
C++ (n_threads = 1)	17.76	0.05	17.81
R/C++	95.06	0.69	237.09
R	112.15	11.50	514.90

Table 3: Speed Comparison probKMA algorithm (`iter_max` = 20): `C++` refers to the second package, `R/C++` to the first package and `R` to the previous `R` implementation.

Table 4

	user	system	elapsed
C++ (n_threads = 7)	158.42	1.46	32.92
C++ (n_threads = 3)	108.55	0.12	38.77
C++ (n_threads = 1)	100.44	0.20	101.41
R/C++	446.71	3.37	1116.13
R	588.56	67.36	3822.11

Table 4: Speed Comparison probKMA algorithm (`iter_max` = 100): `C++` refers to the second package, `R/C++` to the first package and `R` to the previous `R` implementation.

Table 5

	user	system	elapsed
C++ (n_threads = 7)	254.72	1.45	44.94
C++ (n_threads = 3)	163.76	0.19	58.41
C++ (n_threads = 1)	136.94	0.13	137.31
R/C++	597.23	4.29	1498.81
R	835.92	94.55	5812.14

Table 5: Speed Comparison probKMA algorithm (`iter_max` = 200): `C++` refers to the second package, `R/C++` to the first package and `R` to the previous `R` implementation.

The number of threads and the parallelism in executing `find_candidate_motifs` were handled equally to the vector data case. The Table 6 shows the results obtained. To achieve convergence in a reasonable number of iterations, a distance of type L^2 was considered.

Table 6

	user	system	elapsed
C++	1.69	1.92	1268.19
R/C++	2.60	2.45	21789.94
R	8.14	8.42	48988.96

Table 6: Speed Comparison `find_candidate_motifs`: `C++` refers to the second package, `R/C++` to the first package and `R` to the previous `R` implementation.

By rewriting the function `find_candidate_motifs` the results of the second package could still be improved since at the moment to use parallelism in `R` it was necessary to create a function *wrap* in `R` that instantiates each time a new object of class `probKMA` by casting the functional data at each iteration of `find_candidate_motifs`.

In any case, the results obtained allow us to conclude that by finishing rewriting the functions of the previous implementation `R` in `C++`, the time to execute the code even on large data is extremely reduced, allowing the analysis of multivariate and complex functional datasets.

The following Figures 13,14,15 represent the motifs obtained with $K = 3$ and $c = 40$ after the cleaning step performed at the end of the probKMA iterations.

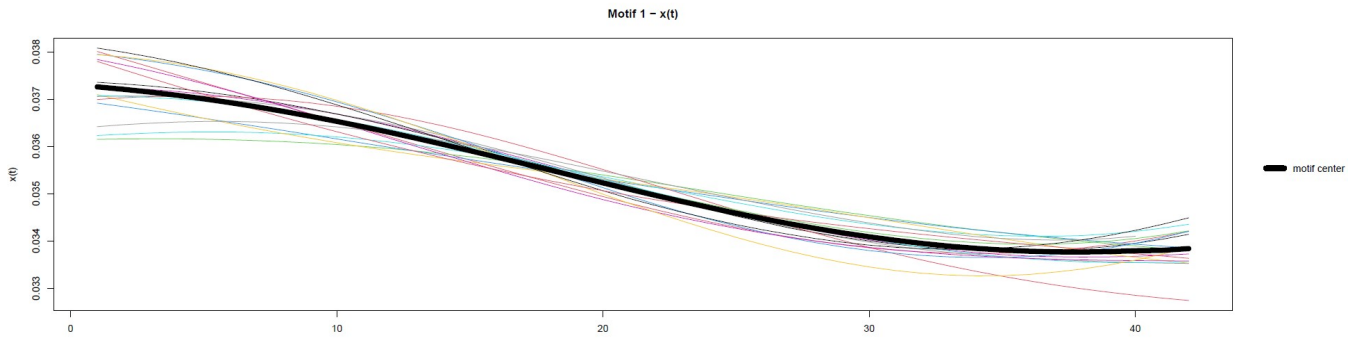

Figure 13: First motif.

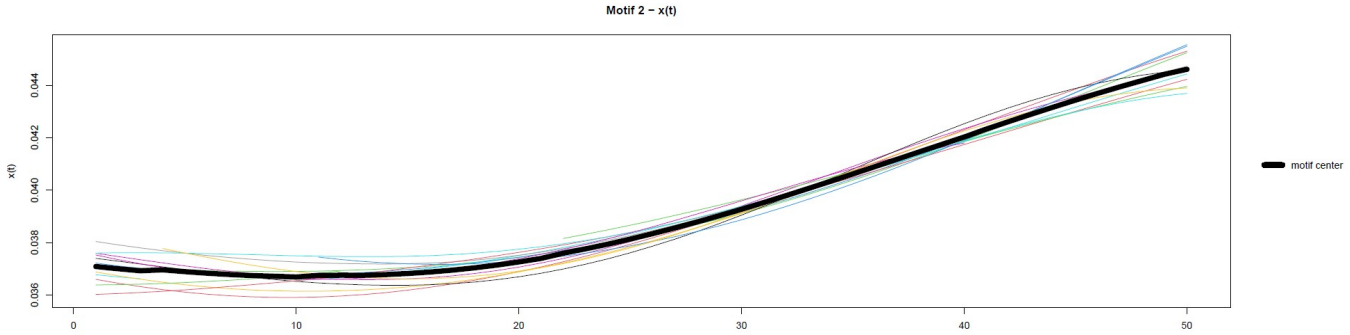

Figure 14: Second motif.

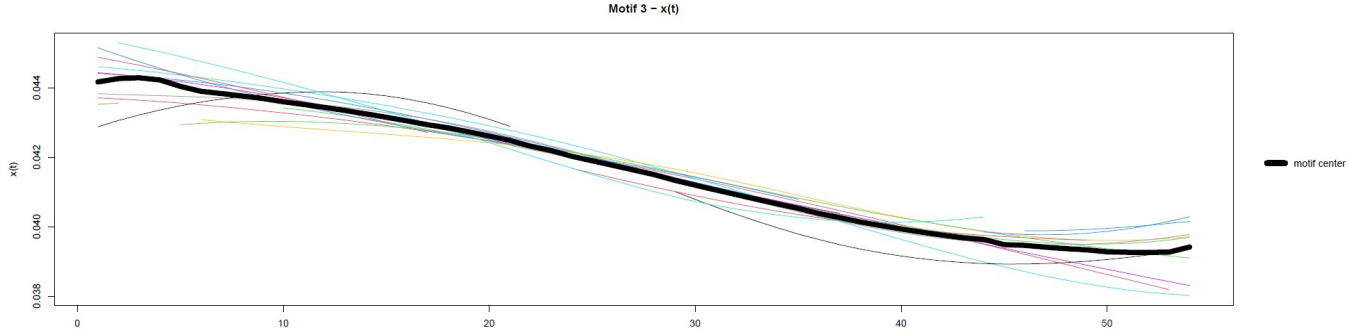

Figure 15: Third motif.

The following future represents for each of the three motifs the probability of each curve belonging to one of the clusters. The darker bars indicate the probabilities that, after dichotomisation, are equal to one, i.e. to which clusters the curves belong according to `P_clean` (`P_clean`$(k, i) = 1$ if and only if curves i assigned to motif k after cleaning).

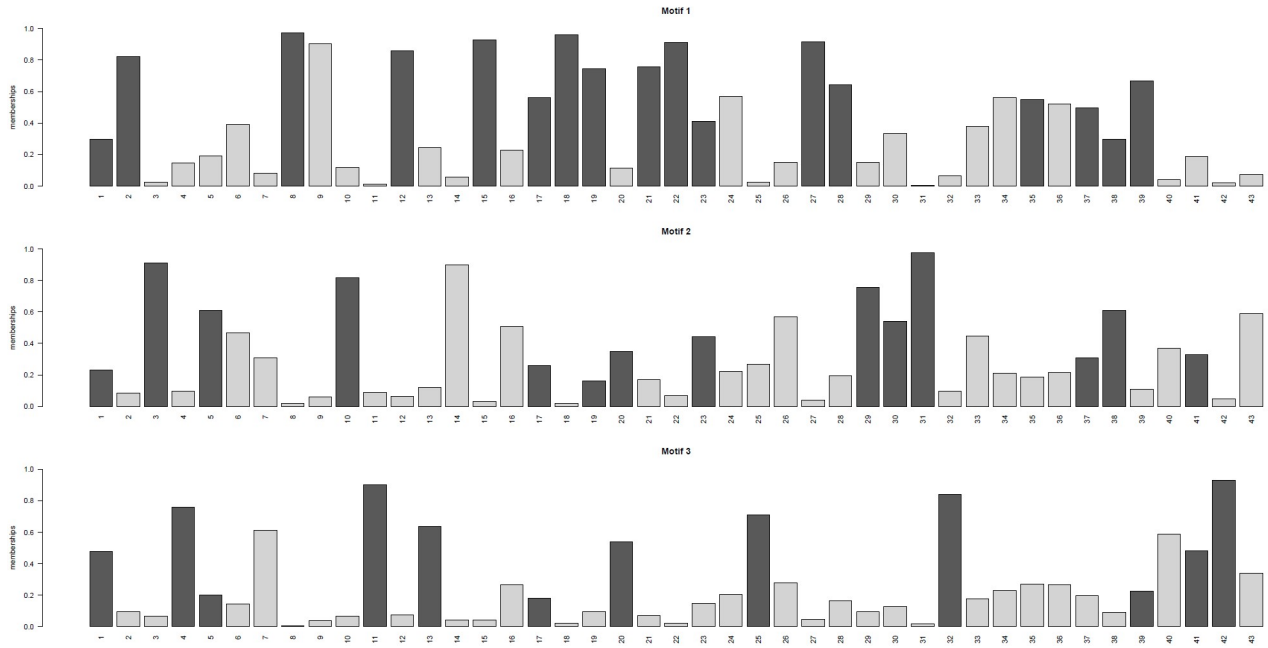

Figure 16: Membership probabilities.

To assess the goodness of the motifs obtained and the corresponding clustering result, the following figure shows the silhouette indexes. For each of the motifs, the average silhouette index is close to one, demonstrating the quality of the result.

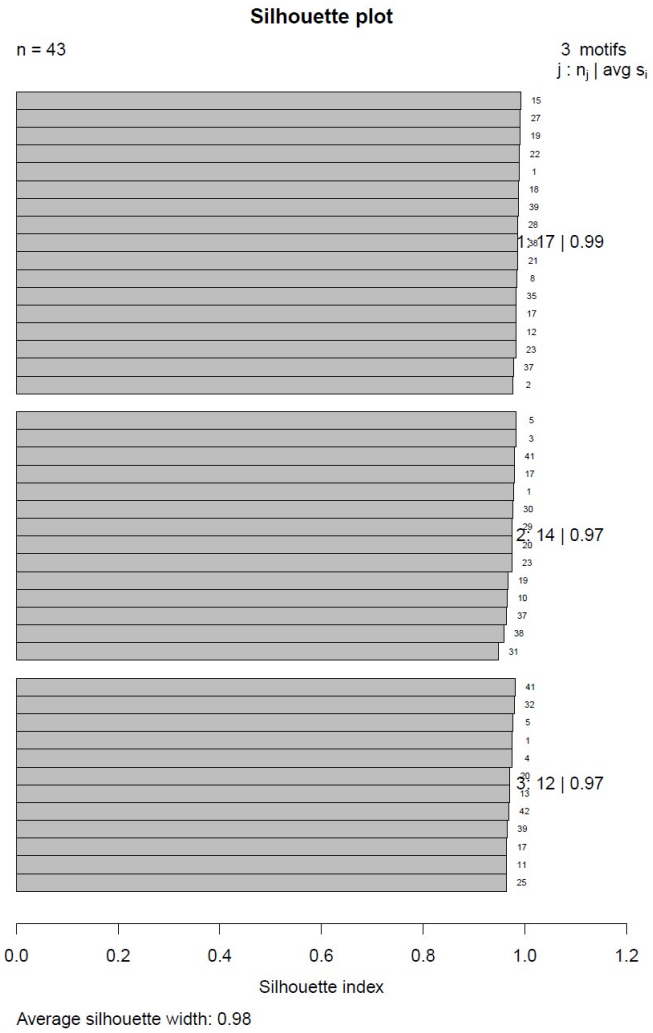

Figure 17: Silhouette indexes plot. For each motif, the bars represent the silhouette index of the curves assigned to that motif. The values on the right are the average silhouette indexes for every motif.

The last remark we want to make is that parallelism in `R` when making extensive use of *Rcpp* can cause problems in rare cases. It may happen that when running `find_candidate_motifs` in the first package, the code breaks due to errors on one of the parallel nodes. Since we fixed the seeds, it is possible to re-launch the code, and it will pick up exactly where it left off with the same initialisation terminating the execution.

6.3. Computational Environment

The simulations were carried out with a Surface Laptop 2, with 8 GB RAM and an Intel $i5-8250U$ processor and within an R framework 4.3.1 and Rstudio $2022.02.3+492$.

7. Conclusions

The two packages implemented respond to different needs. On the one hand, we optimized the code under restrictions due to the use of a pre-existing code in `R` that still needs to be modified even by those who do not know how to use `C++`, while in the case of the second package we implemented a clean code structure that allows the package to be extended clearly and efficiently but using `C++`. The obtained performance shows that the second package is more suitable for high-dimensional data analysis. Therefore, we leave it as future work to finish its implementation, including an adapted version of `probKMA_silhouette` and `motifs_search`. Regarding other possible extensions in the immediate future that the packages should incorporate, we refer to the work by Cremona et al. (2023) [4], which proposes a normalized functional distance to identify functional motifs in stock market prices. Moreover, they also introduce the possibility for the user of providing a priori candidate motif shapes. The idea comes from applications in finance, where data have motifs characterized by

spikes that are difficult to detect in the current version of the code.

7.1. Acknowledgement

We would like to thank Professor Marzia A. Cremona for her help in defining the project and following our work.

8. Practical Questions

8.1. How to install the package?

The package is linked against *OpenMP*, *BLAS* and *LAPACK* libraries in *Makevars*. The package can be installed directly from GitHub. To do this devtools library needs to be installed and loaded in the `R` environment. The dependencies (*Rcpp* and *RcppArmadillo*) will be automatically installed.
The following commands (in `R`) allow you to install the second version of the package:

```
install.packages("devtools")
library(devtools)
install_github('NiccoloF/ProbKMA-FMD',ref = "main_3",subdir = 'ProbKMAcpp')
```

More info can be founded at `https://github.com/NiccoloF/ProbKMA-FMD`.

8.2. How to extend the code?

In this section we can find simple instructions to add new features.

If we want to introduce a new type of dissimilarity, the steps to follow are:

- **Declaration**: In *Dissimilarity.hpp* implement a class derived from `Dissimilarity`, as follows:

```
class newDistance final : public Dissimilarity
{
public:
    virtual double computeDissimilarity(const KMA::Mfield& Y_i,
                                        const KMA::Mfield& V_i)
                                        const override;

    virtual void set_parameters(const Parameters & newParameters)
                                override;

    virtual void computeDissimilarityClean(
        KMA::matrix &D_clean,
        const KMA::imatrix & S_clean,
        const std::vector<arma::urowvec> & V_dom_new,
        const KMA::Mfield & V_clean,
        const KMA::Mfield & Y) const override;

    virtual KMA::vector find_diss(const KMA::Mfield Y,
                                  const KMA::Mfield V,
                                  const KMA::vector& w,
                                  double alpha, unsigned int c_k)
                                  const override;

protected:
      virtual double distance(const KMA::matrix& y,
                              const KMA::matrix& v) const override;

};
```

  As can be seen from the declaration, this class has virtual members that must override the pure virtual methods of the base class;

- **Definition** In a *.cpp* it is necessary to define all pure virtual functions according to the newly introduced distance;
- **Registration** To use the dissimilarity method added it needs to be registered in the `_ProbKMAimp` class, using the factory that can be found in *_ProbKMAimp.cpp*.

```
util::SharedFactory<Dissimilarity> dissfac;
dissfac.FactoryRegister<L2>("L2",_parameters._w);
dissfac.FactoryRegister<H1>("H1",_parameters._w,_parameters._alpha);
dissfac.FactoryRegister<newDistance>("newDistance");
```

The same process can be followed to introduce a new functional or a new way to compute motifs.

References

[1] Marzia A. Cremona and Francesca Chiaromonte. Probabilistic k-means with local alignment for clustering and motif discovery in functional data. *Journal of Computational and Graphical Statistics*, 32(3):1119–1130, 2023.

[2] Dirk Eddelbuettel and Romain Francois. Rcpp: Seamless r and c++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011.

[3] Dirk Eddelbuettel and Conrad Sanderson. Rcpparmadillo: Accelerating r with high-performance c++ linear algebra. *Computational Statistics and Data Analysis*, 71:1054–1063, March 2014.

[4] Federico Severino Marzia A. Cremona, Lyubov Doroshenko. Functional motif discovery in stock market prices. *SSRN*, 2023.

[5] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[6] Laura M. Sangalli, Piercesare Secchi, Simone Vantini, and Valeria Vitelli. k-mean alignment for curve clustering. *Computational Statistics Data Analysis*, 54(5):1219–1233, 2010.

[7] Hadley Wickham, Peter Danenberg, Gábor Csárdi, and Manuel Eugster. *roxygen2: In-Line Documentation for R*, 2024. R package version 7.3.1, https://github.com/r-lib/roxygen2.