`flm`: the FLAME method for estimation and variable selection in Function-on-Scalar regression problems

Alice Parodi and Matthew Reimherr

2017-10-05

Contents

1 Introduction

The `flm` package provides an efficient tool to deal with Function-on-Scalar regression problems, mainly when the number of predictors `I` is much larger than the number of statistical unit `N`. The main function of `flm` is `FLAME` that detects the set of significant predictrs and estimates their coefficients with the FLAME method. FLAME, *functional linear adaptive mixed estimation* is a methodology that simultaneously exploits the smoothness of the functional parameters as well as the sparsity of the predictors.

The Function-on-Scalar regression problem that FLAME aims to solve is

$$Y_n = \sum_{i=1}^{I} X_{n,i}\beta_i^\star + \varepsilon_n,$$

where $Y_1, \dots, Y_N$ are independent random elements of a general Hilbert space $\mathbb{H}$, $\mathbf{X} = \{X_{n,i}\} \in \mathbb{R}^{N\times I}$ is a deterministic design matrix with standardized columns and ε_n are i.i.d. Gaussian random elements of $\mathbb{H}$ such that ε_n have 0 mean and covariance operator C.

Then, given the response functions and the predictors, the function `FLAME` can automatically identify the significant predictors and define the coefficients in a proper RKHS.

In Section 2 and 3 the detailed procedure of the estimation, from the definition of the kernel of the `generation_kenrel` and `generation_kernel_periodic` functions, to the solution of the Function-on-Scalar regression problem of the `estimation_beta` function, with some details on the algorithm implementation. In Section 4, instead, an exemple of an automatical usage of the package through the introduction of the `FLAME` function.

2 Defintion of the kernel

The main advantage of FLAME is the possibility of controlling the smoothness of the parameter estimates with the definition of a proper reproducing kernel Hilbert space.

`flm` has two functions to define different RKHSs: given a specific kernel K, `generation_kernel` and `generation_kernel_periodic` define its eigenvalues θ_j, eigenfunctions v_j and their derivatives. Then K becomes, for the spectral theorem [Dunford and Schwartz, 1963]

$$K = \sum_{j=1}^{\infty} \theta_j v_j \otimes v_j$$

The kernel we examine in this package are the Sobolev, the Exponential, the Gaussian (Section 2.1) and the Periodic kernel (Section 2.2).

2.1 Exponential, Sobolev and Gaussian kernel

The `generation_kernel` function allows the user to define the Exponential, the Sobolev and the Gaussian kernel.

Here an explicit definition of the three kernels:

- **Sobelev kernel**: Consider $\mathbb{H} = L^2(\mathcal{D})$, where $\mathcal{D}$ is a compact subset of $\mathbb{R}^d$. We can define $\mathbb{K}$ to be the subset of functions in $L^2(\mathcal{D})$ that have up to and including m^{th} order derivatives that are also in $L^2(\mathcal{D})$. In this package we limit our analysis to $m = 1$ and $d = 1$. Then, we define a family of norms on $\mathbb{K}$ as

  $$\|x\|_{\mathbb{K}}^2 = \int_{\mathcal{D}} |x(s)|^2 \, ds + \frac{1}{\sigma} \int_{\mathcal{D}} |x^{(\prime)}(s)|^2 \, ds;$$

  here the σ parameter controls the influence of the H^1 norm and then the smoothness of the eigenfunctions. Increasing σ the smoothness level decreases. Equipped with this norm, $\mathbb{K}$ is an RKHS if and only if $m > d/2$, as in our case of one-dimensional functions ($d = 1$) in H^1 ($m = 1$). The kernel cannot always be written down explicitly, but in the case where $\mathcal{D} = [0, 1]$ and $m = 1$, we have that

  $$K(t, s) = \begin{cases} \frac{\sigma}{\sinh(\sigma)} \cosh(\sigma(1 - s)) \cosh(\sigma t) & t \leq s \\ \frac{\sigma}{\sinh(\sigma)} \cosh(\sigma(1 - t)) \cosh(\sigma s) & t > s \end{cases} .$$

  Then we can numerically solve the equation to isolate the eigenfunctions and the eigenvalues of K as the `sobolev_kernel` function does. This function is implicitly called in `generation_kernel`. Details on the Sobolev kernel can be found in [Berlinet and Thomas-Agnan, 2011].

- **Gaussian Kernel** Let $\mathbb{H} = L^2(\mathcal{D})$, with $\mathcal{D}$ a compact subset of $\mathbb{R}^d$. The Gaussian kernel if $d = 1$ is given by

  $$K(s, s') = \exp\left\{-\sigma |s - s'|^2 .\right\}.$$

  While the Sobolev spaces contain functions which are differentiable up to a given order, the space $\mathbb{K}$ here contains functions which are infinitely differentiable. When used in FLAME, such a kernel produces very smooth estimates. As for the Sobolev kernel, the smoothness level of the kernel is controlled by the σ parameter. Increasing σ the smoothness level is reduced and FLAME get a more rough estimates. The definition of the kernel function is coded in the `kernlab` R package [Karatzoglou et al., 2004].

- **Exponential Kernel**: The exponential kernel is on the other end of the "smoothness" spectrum compared to the Gaussian kernel. In the one-dimensional case we have

  $$K(s, s') = \exp\left\{-\sigma |s - s'|\right\}.$$

This seemingly minor adjustment to the power in the exponent produces a space consisting of continuous functions which need not to be differentiable.
Using this kernel produces substantially rougher FLAME estimates than the Gaussian kernel. They are also a bit rougher than the Sobolev kernel as well. As for the previous kernels, the smoothness parameter σ tunes the regularity level of the FLAME estimations. And as for the Gaussian kernel the `kernlab` R package [Karatzoglou et al., 2004] provides an explicit definition of the kernel matrix.

The `generation_kernel` function, then, allows the user to define the eigenfunctions and eigenvalues of these three different kernels, once the time domain is defined in the `domain` argument. The `type_kernel` parameter defines the type of kernel: `'exponential'`, `'sobolev'` and `'gaussian'` are the three possible choices; the `param_kernel` argument, instead, is the σ parameter tuning the regularity level. The number of eigenfunctions v_j (which define the basis functions of the RKHS) is chosen as

$$\sum_{j=1}^{J} \theta_j \geq \text{thres} \sum_{j=1}^{\infty} \theta_j.$$

where the `thres` parameter is an input of the `generation_kernel` function and θ_j are the eigenvalues of the kernel.

In the following chunk an example of definition of Sobolev kernel with $\sigma = 8$ and in **??** the first four eigenfunctions and their derivatives, with the correspondent ratio of explained variability $\theta_j / \sum_j \theta_j$.

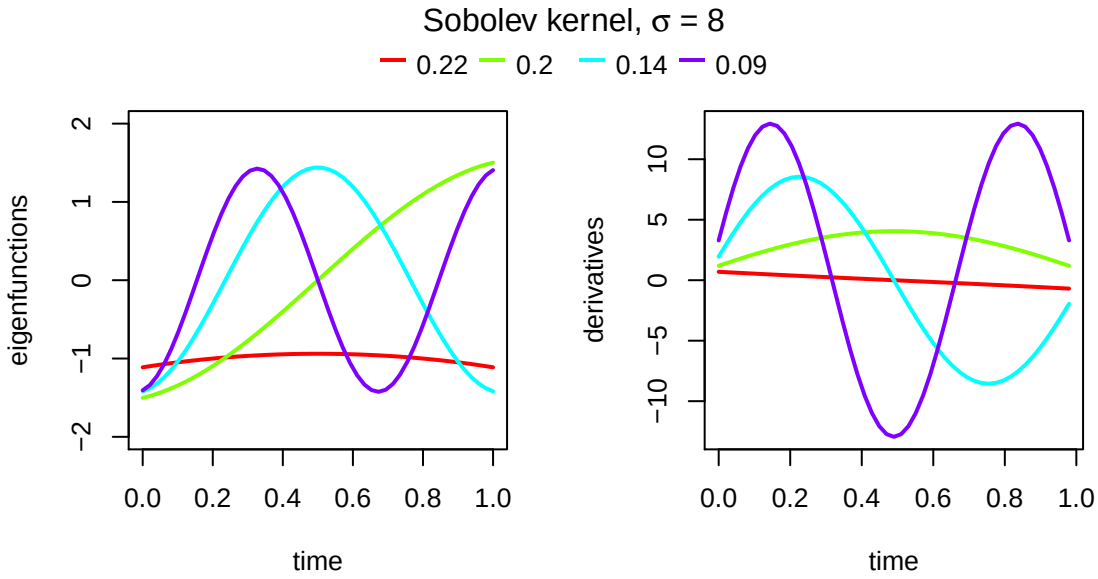
```
type_kernel <-  'sobolev'
param_kernel <-  8
M <- 50
T_domain <-  seq(0, 1, length = M) # time point grid.
thres <- 0.99 # thresold for the eigenvalues.
kernel_here <- generation_kernel(type = type_kernel,
                                 parameter = param_kernel,
                                 domain = T_domain,
                                 thres = 0.99,
                                 return.derivatives = TRUE)
eigenval <- kernel_here$eigenval
eigenvect <- kernel_here$eigenvect
derivatives <- kernel_here$derivatives

layout(mat = matrix(c(1,2,1,3),  nrow = 2, ncol = 2),
       heights= c(0.6,3.5),respect = TRUE, widths = c(4,4))
# legend
par(oma=c(0, 0, 0, 0), mar =c(0,0,2,0))
plot(c(1,2,3),c(0,1,1),col=0, xlab ='', ylab='', axes = FALSE)
legend('center',  legend = round(eigenval[1:4]/sum(eigenval), 2),
       xpd = TRUE, horiz = TRUE,
       inset = c(0,  0.98), bty = "n", pch = '-', pt.cex = 2,
       col = rainbow(4), cex = 1)
title( main = expression(paste('Sobolev kernel, ', sigma, ' = 8')),
       cex.main = 1.2, font.main = 2)

# plot of the eigenfunctions
par(mar=c(4,4, 1, 2) + 0.1)
matplot(T_domain, eigenvect[,1:4], type = 'l', lwd = 2, xlab = 'time',
        ylab ='eigenfunctions',
        lty = 1, cex = 1, cex.main = 1, cex.lab = 1,
        cex.axis = 1, col =rainbow(4), ylim = c(-2,2))

# plot of the derivatives
```

```
par(mar=c(4,4,1, 2) + 0.1)
matplot(T_domain[-50], derivatives[,1:4], type = 'l', lwd = 2, xlab = 'time',
        ylab ='derivatives',
        lty = 1, cex = 1, cex.main = 1, cex.lab = 1,
        cex.axis = 1, col =rainbow(4))
```



This example of Sobolev kenrel is stored in the `SobolevKernel` dataset of the `flm` package.

2.2 Periodic kernel

A very useful feature of working with an RKHS is that one can also include periodicity and boundary conditions into the parameter estimates, using the `generation_kernel_periodic` function, for example, you can define a kernel with a fixed periodicity p and a smoothing parameter σ. If you have yearly measurements with seasonal or semestral periodicity, for example, you may use the periodic kernel with period $p = 1/4$ or $p = 1/2$.

The kernel for period p on a one dimensional dimanin is defined as

$$K(s, s') = \sigma^2 \exp\left\{-2/\sigma \sin^2\left(\frac{\pi|s - s'|}{p}\right)\right\}.$$

In the following chunk an example of definition of periodic kernel with `period = 1/2` and in Figure 1 the first four eigenfunctions and their derivatives, with the correspondent ratio of explained variability.

```
kernel_here <- generation_kernel_periodic(period = 1/2,
                                          parameter = param_kernel,
                                          domain = T_domain,
                                          thres = 1-10^{-16},
                                          return.derivatives = TRUE)
eigenval <- kernel_here$eigenval
eigenvect <- -kernel_here$eigenvect
derivatives <- kernel_here$derivatives

layout(mat = matrix(c(1,2,1,3),  nrow = 2, ncol = 2),
       heights= c(0.6,3.5),respect = TRUE, widths = c(4,4))
# legend
par(oma=c(0, 0, 0, 0), mar =c(0,0,2,0))
```

```r
plot(c(1,2,3),c(0,1,1),col=0, xlab ='', ylab='', axes = FALSE)
legend('center',  legend = round(eigenval[1:4]/sum(eigenval), 3),
       xpd = TRUE, horiz = TRUE,
       inset = c(0,  0.98), bty = "n", pch = '-', pt.cex = 2,
       col = rainbow(4), cex = 1)
title( main = expression(paste('Periodic kernel, period = 1/2')),
       cex.main = 1.2, font.main = 2)

# plot of the eigenfunctions
par(mar=c(4,4, 1, 2) + 0.1)
matplot(T_domain, eigenvect[,1:4], type = 'l', lwd = 2, xlab = 'time',
        ylab ='eigenfunctions',
        lty = 1, cex = 1, cex.main = 1, cex.lab = 1,
        cex.axis = 1, col =rainbow(4), ylim = c(-2,2))

# plot of the derivatives
par(mar=c(4,4,1, 2) + 0.1)
matplot(T_domain[-50], derivatives[,1:4], type = 'l', lwd = 2, xlab = 'time',
        ylab ='derivatives',
        lty = 1, cex = 1, cex.main = 1, cex.lab = 1,
        cex.axis = 1, col =rainbow(4))
```

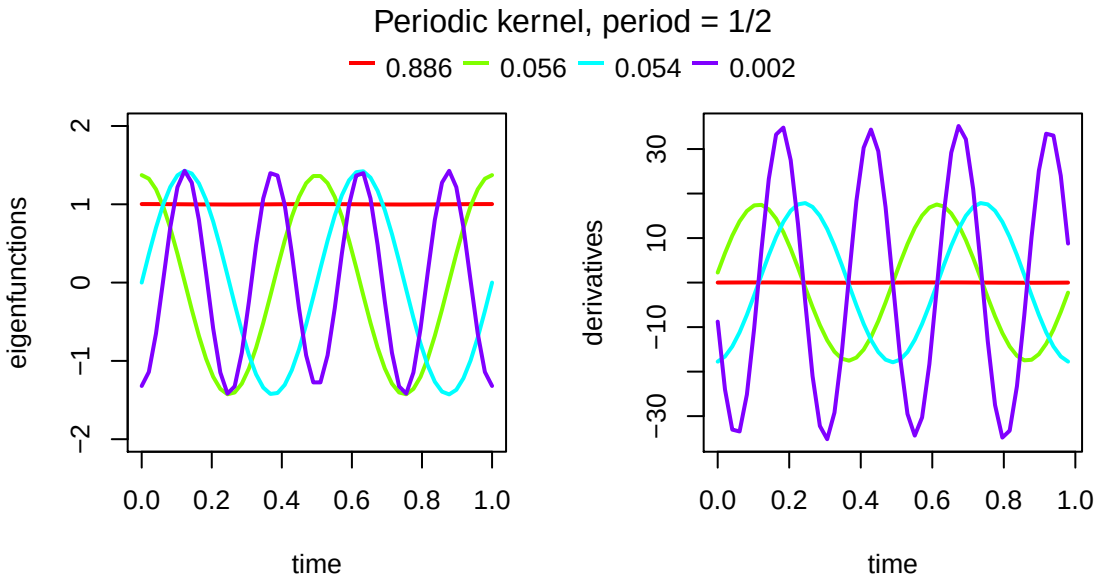


Figure 1: Plot of the first 4 eigenfunctions (left panel) and derivatives (right panel) of the periodic kernel with period $p = 1/2$. The correspondent explained variance is on the top of the plot.

3 FLAME estimation

In this section we define an example of generation of data for a Function-on-Scalar linear model (Section 3.1), we present the definition of the kernel for the estimation (Section 3.2) and an outline of the FLAME method (Section 3.3) with an analysis of the results (Section 3.4).

3.1 Generation of data

We define an high-dimensional setting simulation with `N = 500` and `I = 1000` to highlight both the efficiency of FLAME in the estimation and in variable selection. Only `I0 = 10` predictors, in fact, are meaningful for the response, the others have null effect on the Y's.

The predictor matrix $\mathbf{X}$ is the standardized version of a matrix randomly sampled from a N dimension Gaussian distribution with 0 average and covariance C. The true coefficients $\beta^{\star}(t)$ are sampled from a Matern process with 0 average and parameters $(\nu = 2.5, \text{range} = 1/4, \sigma^2 = 1)$.

Observations $y(t)$ are, then, obtained as the sum of the contribution of all the predictors and a random noise, a 0-mean Matern process with parameters $(\nu = 1.5, \text{range} = 1/4, \sigma^2 = 1)$. Functions are sampled on a $m = 50$ points grid.

The Matern covariance operator is defined in the `covMaterniso` function.

In Figure 2 the plot of the coefficients $\beta^{*}(t)$, 20 random errors $\varepsilon(t)$ and the correspondent response functions $Y(t)$.

```
N <-  500
I <- 1000
I0 <- 10

# defintion of the time domain
m <- 50 # total number of points
T_domain <-  seq(0, 1, length = m) # time points, length = m
M_integ <- length(T_domain)/diff(range(T_domain)) # coefficient for the
# computation of the integrals

# defintion of the design matrix X, in this specific case the
# covariance matrix C is the identity matrix
mu_x <- rep(0, I)
C <- diag(I)
X <-  mvrnorm(n=N, mu=mu_x, Sigma=C)
X <- scale(X) # normalization

# defintion of the coefficients
nu_beta <- 2.5
range <-1/4
variance <- 1
hyp <- c(log(range), log(variance)/2) # set of parameters for the
# Matern Covariance operator of beta
mu_beta <- rep(0,m) # mean of the beta
Sig_beta <- covMaterniso(nu_beta, rho = range, sigma = sqrt(variance) , T_domain)
beta <- mvrnorm(mu=mu_beta, Sigma=Sig_beta, n=I0) # generation of the
# I0 significant coefficients

# defintion of the random errors
nu_eps <- 1.5
mu_eps <- rep(0, m)
Sig_eps <- covMaterniso(nu_eps, rho = range, sigma = sqrt(variance), T_domain)
eps <- mvrnorm(mu=mu_eps, Sigma=Sig_eps, n=N) # generation of the N
# random errors

I_X <- sort(sample(1:I, I0)) # index of the I0 significant predictors
```

```
Y_true <- X[,I_X]%*%beta
Y_full <- X[,I_X]%*%beta + eps # Y_n observations

par(mfrow = c(1,3), mar=c(4,4,3,2) + 0.1)

# plot of the true beta coefficients
matplot(T_domain, t(beta), type = 'l', lwd = 2, xlab = 'time',
        ylab ='beta',
        lty = 1, cex = 2,  cex.lab = 1.5,
        cex.axis = 1.5, col =rainbow(10))
title(main = expression(paste(beta, '*',(t),' coefficients', sep ='')),
      cex.main = 1.5, font.main = 2)

# plot of the errors
matplot(T_domain, t(eps[1:20,]), type = 'l', lwd = 2, xlab = 'time',
        ylab ='beta',
        lty = 1, cex = 2,  cex.lab = 1.5,
        cex.axis = 1.5, col =rainbow(20))
title(main = expression(paste(epsilon(t),' coefficients', sep ='')),
      cex.main = 1.5, font.main = 2)

# plot of the response functions
matplot(T_domain, t(Y_full[1:20,]), type = 'l', lwd = 2, xlab = 'time',
        ylab ='beta',
        lty = 1, cex = 2, cex.lab = 1.5,
        cex.axis = 1.5, col =rainbow(20))
title(main = expression(paste(Y(t),' functions', sep ='')),
      cex.main = 1.5, font.main = 2)
```

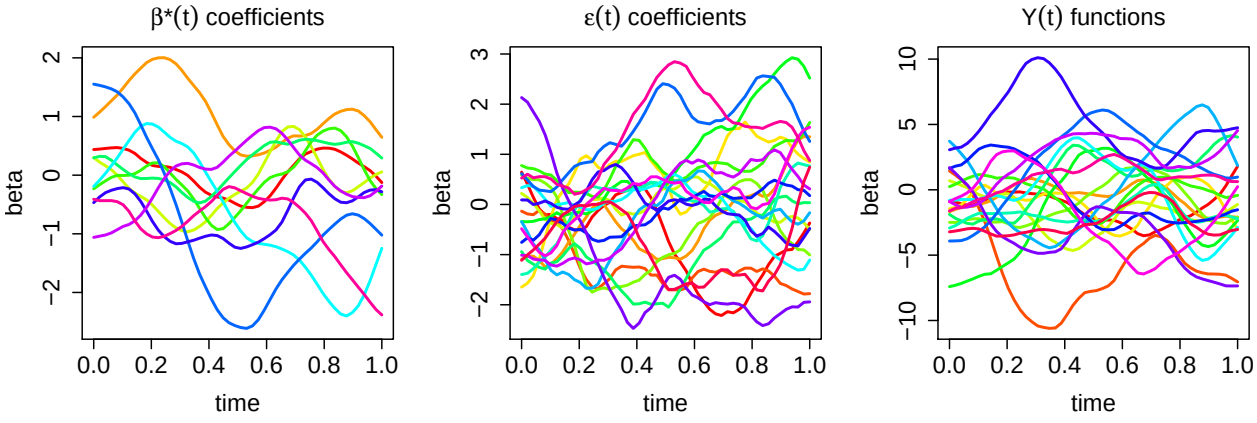


Figure 2: Random generation of data. From the left, 10 coefficients $\beta^*(t)$, 20 random errors $\varepsilon(t)$ and the correspondent 20 response functions $Y_n(t)$.

This data are stored in the `simulation` dataset of `flm`.

3.2 Defintion of the kernel and projection

For the simulation we are running we choose as kernel the Sobolev kernel with $\sigma = 8$ and a threshold for the eigenvalues 0.99. The eigenfunctions of the kernel are an orthogonal basis both for the space $\mathbb{H}$ and for $\mathbb{K}$;

then for the following estimation we can project the $Y_n(t)$ functions on that basis with the `projection_basis` function.

```
# defintion of the kernel
type_kernel <-  'sobolev'
param_kernel <-  8
m <- 50
T_domain <-  seq(0, 1, length = m) # time point grid.
thres <- 0.99 # thresold for the eigenvalues.
kernel_here <- generation_kernel(type = type_kernel,
                                 parameter = param_kernel,
                                 domain = T_domain,
                                 thres = 0.99,
                                 return.derivatives = TRUE)
eigenval <- kernel_here$eigenval
eigenvect <- kernel_here$eigenvect
derivatives <- kernel_here$derivatives

# preojection on the kernel basis of y and beta
Y_matrix <- projection_basis(Y_full, eigenvect, M_integ)
B_true <- projection_basis(beta, eigenvect, M_integ)

matrix_beta_true_full <- matrix(0, dim(B_true)[1], I)
matrix_beta_true_full[,I_X] <- B_true
```

Given the definition of the derivatives of the eigenfunctions of the kernel (returned by the `generation_kernel` function), we can also define the derivatives of the true coefficients β^* and of the responses

```
B_true_der <- t(kernel_here$derivatives %*% B_true)

Y_true_der <- X[,I_X]%*%B_true_der
```

3.3 FLAME estimation

The function `estimation_beta` is the main function of the `flm` package and performs the FLAME estimation. The back-end of this function is written in `c++` (in the `FLAME_functions_cpp.cpp` function), so that the computation is efficient also in the high dimensional setting.

The function mainly consist of a coordinate-descent algorithm to define the FLAME estimation minimizing the target function

$$L(\beta) = \frac{1}{2N}\sum_{n=1}^{N}||Y_n - X_n^\top\beta||_{\mathbb{H}}^2 + \lambda\sum_{i=1}^{I}\tilde{\omega}_i||\beta_i||_{\mathbb{K}} = \frac{1}{2N}||Y - \mathbf{X}\beta||_{\mathbb{H}}^2 + \lambda\sum_{i=1}^{I}\tilde{\omega}_i||\beta_i||_{\mathbb{K}}$$

with $Y \in \mathbb{H}^N$, $\mathbf{X} \in \mathbb{R}^{N\times I}$ and $X_n = \mathbf{X}_{(n,\cdot)} \in \mathbb{R}^I$, $\beta \in \mathbb{K}^I$. Throughout, we use notation such as $\mathbb{H}^N$ to denote product spaces. For the sake of simplicity, we abuse notation by letting $\|\cdot\|_{\mathbb{H}}$ also denote the induced Hilbert space norm on product spaces such as $\mathbb{H}^N$.

The $\tilde{\omega}_i$ parameters are used to balance the contribution of the different coefficients and to make the LASSO estimator unbiased. The function `estimation_beta` has the estimation of $\tilde{\omega}_i$ as first objective. The coordinate-descent algorithm, in fact is run twice. The first one, the *non adaptive step*, is run defining as 1 all the weights $\tilde{\omega}_i$ and the second one, the *adaptive step*, is run, to obtain an unbiased estimator, with

$$\tilde{\omega}_i = \frac{1}{\|\hat{\beta}_i^1\|_{\mathbb{K}}},$$

where $\hat{\beta}_i^1$ the estimated coefficient of the *non-adaptive step*.

A key parameter for the estimation is λ, used to balance the prediction error $||Y - \mathbf{X}\beta||_{\mathbb{H}}^2$ and the smoothness level of the estimations $\sum_{i=1}^{I} \tilde{\omega}_i ||\beta_i||_{\mathbb{K}}$. The two steps of the algorithm are both run on a grid of λ and the best value is chosen with a cross-validation criteria, selecting a training set, made up by the `proportion_training_set` percent of the data, and the remaining test set. The `estimation_beta` function automatically defines the grid for the λ parameter in the two runs as a logarithmic equispaced gird from a maximum value, $\lambda_{\max}$

$$\lambda_{\max} = \max_{i=1,\dots,I} \omega_i^{-1} \|N^{-1} \sum X_{ni} K(Y_n)\|_{\mathbb{K}}$$

to the minimum value `ratio_lambda` $\cdot \lambda_{\max}$. The user, beside the `ratio_lambda` parameter can define also the length of the grid, in the `number_lambda` parameter.

Focusing on the coordinate-descent method. It is based on the subgradient equation

$$\begin{aligned}
\frac{\partial}{\partial \beta_i} L(\beta) &= -\frac{1}{N} \sum_{n=1}^{N} X_{n,i} K(Y_n - X_n^\top \beta) + \lambda \tilde{\omega}_i \begin{cases} ||\beta_i||_{\mathbb{K}}^{-1} \beta_i, & \beta_i \neq 0 \\ \{h \in \mathbb{K} : ||h||_{\mathbb{K}} \leq 1\}, & \beta_i = 0 \end{cases} \\
&= -K(\tilde{\beta}) + K(\beta_i) + \lambda \omega_i \begin{cases} ||\beta_i||_{\mathbb{K}}^{-1} \beta_i, & \beta_i \neq 0 \\ \{h \in \mathbb{K} : ||h||_{\mathbb{K}} \leq 1\}, & \beta_i = 0 \end{cases}.
\end{aligned}$$

with $\tilde{\beta}$ the least squares estimator $\tilde{\beta}_i = \frac{1}{N} \sum_{n=1}^{N} X_{n,i} E_n$ where E_n is the residual $E_n = Y_n - \sum_{j \neq i} X_{n,j} \hat{\beta}_j$ and it is updated at each iteration. From the subgradient equation we can also detect the meaning of the maximum value for λ: $\lambda_{\max}$, in fact, is the minimum value of λ for which all the predictors are guarantee to have 0 coefficient. For all i

$$||K(\tilde{\beta}_i)||_{\mathbb{K}} \leq \lambda \omega_i$$

```
time <- proc.time()
FLAME <- estimation_beta(X = X, # design matrix
                         Y = Y_matrix, # response functions projected on the kernel basis
                         eigenval = eigenval, # basis
                         NoI = 10, # max. num. iterations coordinate descent
                         thres = 0.1, # stopping threshold for the coordinate descent
                         number_non_zeros = I0*2, # kill switch parameter
                         ratio_lambda = 0.01, # ratio for the min. lambda
                         number_lambda = 100, # num. elements of the grid for lambda
                         proportion_training_set = 0.75, # training set
                         verbose = FALSE) # no show of all the iterations
duration <- proc.time()-time
duration
```

```
##    user  system elapsed
##  28.007   0.401  28.755
```

Moreover we present two features we have included in the algorithm to increase the computational efficiency. The first is a *warm start* which means that when moving to the next λ in the grid, we use the previous $\hat{beta}$ as initial estimation and, due to the small changes in λ, this means that the new $\hat{\beta}$ can be computed very quickly. The second feature is the *kill switch* parameter. This allows the user to set the maximum number of significant predictors to be selected by the model: when the algorithm moves past this threshold, the algorithm is stopped.

The `estimation_beta` function automatically performs all this steps and returns both the final result after the *adaptive step* and the intermediate result, just after the *non adaptive step*:

```
names(FLAME)
```

```
## [1] "beta"                    "beta_no_adaptive"
## [3] "predictors"              "predictors_no_adaptive"
```

To directly access to the coordinate descent method and perform manually the estimation, for example fixing a specific value for λ, a specific set of weights or a specific starting point for the estimated β, the user can run the `definition_beta` function, the one that is implicitly called in `estimation_beta`.

We can notice that the `estimation_beta` function returns as the estimation of the coefficients $\hat{\beta}$ the matrix of their projection on the kernel basis. The function `projection_domain` allows to compute the estimation on the time domain and then to represent the results, as in Figure 3. Here we show a comparison with the true simulated β^* functions.

```
beta_on_time_grid <- projection_domain(FLAME$beta, eigenvect)
y_on_grid_estimated <- X %*% beta_on_time_grid

par(mfrow = c(1,2), mar=c(4,4,3,2) + 0.1)

# plot of the true beta coefficients
matplot(T_domain, t(beta), type = 'l', lwd = 2, xlab = 'time',
        ylab ='beta',
        lty = 1, cex = 1,  cex.lab = 1,
        cex.axis = 1, col =rainbow(10), ylim = c(-3, 2))
title(main = expression(paste(beta, '*', ' coefficients', sep ='')),
      cex.main = 1.2, font.main = 2)

# plot of the estimated beta
matplot(T_domain, t(beta_on_time_grid[FLAME$predictors,]), type = 'l',
        lwd = 2, xlab = 'time',
        ylab ='beta',
        lty = 1, cex = 1,  cex.lab = 1,
        cex.axis = 1, col =rainbow(10), ylim = c(-3, 2))
title(main = expression(paste('estimated ',beta,' coefficients', sep ='')),
      cex.main = 1.2, font.main = 2)
```

3.4 Analysis of the results

To analyse the result of this simulation, first of all we detect the relevant predictors isolated by FLAME and we compare them with the true ones and we notice that FLAME correctly isolates the `I0 = 10` relevant predictors, without any false positive predictor added.

```
I_X
```

```
##  [1]  18  97 287 297 433 527 642 709 901 934
```

```
FLAME$predictors
```

```
##  [1]  18  97 287 297 433 527 642 709 901 934
```

```
true_positives <- length(which(I_X %in% FLAME$predictors))
true_positives # number of significant predictors correctly identified
```

```
## [1] 10
```

```
false_positives <- length(which(!(FLAME$predictors %in% I_X)))
false_positives # number of non significant predictors wrongly picked by the algorithm
```

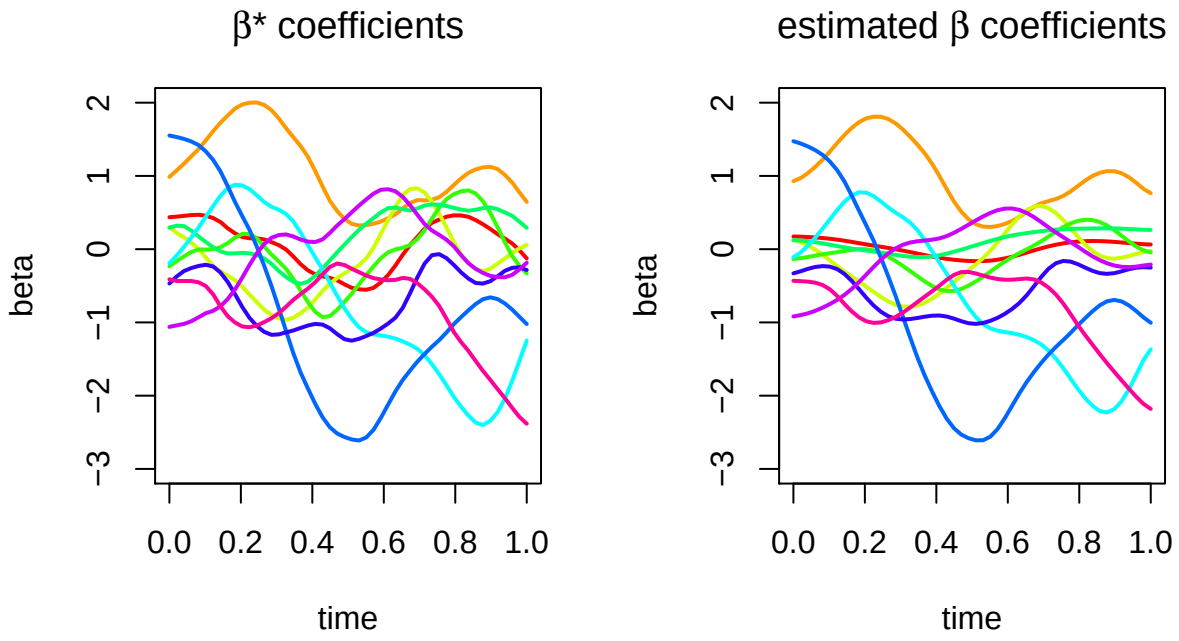


Figure 3: In the left panel the plot of the simulated β^* coefficients, while in the right panel the FLAME coefficients $\hat{\beta}$ are shown.

```
## [1] 0
```

Then we introduce a short analysis of the result computing:

- the prediction error on data $\sum_{n=1}^{N} \| \mathbf{X}_n\beta^* - \mathbf{X}_n\hat{\beta} \|_{L^2}$,
- the prediction error on derivatives $\sum_{n=1}^{N} \| \mathbf{X}_n\beta^{*\prime} - \mathbf{X}_n\hat{\beta}' \|_{L^2}$,
- the $\mathbb{K}$-norm of the error in the prediction of the β coefficients $\sum_{i=1}^{I} \|\beta_i^* - \hat{\beta}_i\|_{\mathbb{K}}$. This last error can be easily computed with the `norm_matrix_K` function.

```
beta_der_on_grid_estimated<- kernel_here$derivatives %*% FLAME$beta

prediction_error <- sum(apply(Y_true - y_on_grid_estimated,
                              1,
                              function(x)
                              {
                               sqrt((2*sum(x^2)-x[1]^2-x[length(x)]^2)/(M_integ*2))
                              }
                              )
                        )
prediction_error
```

```
## [1] 218.5814
```

```
estimated_y_der_grid <- X %*% t(beta_der_on_grid_estimated)
prediction_error_der <- sum(apply(Y_true_der - estimated_y_der_grid,
                                  1,
                                  function(x)
                                  {
                                   sqrt((2*sum(x^2)-x[1]^2-x[length(x)]^2)/((M_integ-1)*2))
                                  }
                                  )
                            )
```

```
prediction_error_der
```

```
## [1] 2688.182
```

```
norm_K_beta <- sum(norm_matrix_K(matrix_beta_true_full - FLAME$beta, eigenval)^2)
norm_K_beta
```

```
## [1] 0.769599
```

4 The automatical usage of the FLAME method to solve Function-on-Scalar regression problems

In this final Section we present the `FLAME` function that allows the user a direct solution of the regression problem. From an `fd` object, or a point-wise evaluation of the response functions and the set of predictors, the function automatically detects the significant predictors and computes the estimation. It is possible to provide the kernel, choosing among the `exponential`, the `gaussian`, the `sobolev` and the `periodic` kernel and fixing the smoothness parameter. Here an example of estimation with the predictors provided as an `fd` object. The $y(t)$ of Section 3.1 are represented as their projection on a 20 elements cubic Bspline basis and then also the estimated coefficients are returned as an `fd` object.

```
class(Y_fd)
```

```
## [1] "fd"
```

```
estimation_auto <-  FLAME(Y_fd, # fd object for the response
                          X, # predictors matrix
                          number_non_zeros = 20)
# default choice for the kernel is Sobolev with sigma = 8,
names(estimation_auto)
```

```
## [1] "beta"       "predictors"
```

```
class(estimation_auto$beta)
```

```
## [1] "fd"
```

```
estimation_auto$predictors
```

```
##  [1]  18  97 287 297 433 527 642 709 901 934
```

References

A. Berlinet and C. Thomas-Agnan. *Reproducing kernel Hilbert spaces in probability and statistics*. Springer Science and Business Media, 2011.

N. Dunford and J. Schwartz. *Linear operators. Part 2: Spectral theory. Self adjoint operators in Hilbert space*. Interscience Publishers, 1963.

Alexandros Karatzoglou, Alex Smola, Kurt Hornik, and Achim Zeileis. kernlab: an s4 package for kernel methods in r. *Journal of Statistical Software*, 11(9):1–20, 2004. URL http://www.jstatsoft.org/v11/i09/.