funGp: An R Package for Gaussian Process Regression with Scalar and Functional Inputs

José Daniel Betancourt, François Bachoc, Thierry Klein, Déborah Idier, Jérémy Rohmer, Yves Deville

funGp: An R Package for Gaussian Process Regression with Scalar and Functional Inputs

José Betancourt
IMT-Université de Toulouse
ENAC

François Bachoc
IMT-Université de Toulouse

Thierry Klein
ENAC
IMT-Université de Toulouse

Déborah Idier
BRGM

Jérémy Rohmer
BRGM

Yves Deville
Alpestat

Abstract

This article introduces **funGp**, an R package which handles regression problems involving multiple scalar and/or functional inputs, and a scalar output, through the Gaussian process model. This is particularly of interest for the design and analysis of computer experiments with expensive-to-evaluate numerical codes that take as inputs regularly sampled time series. Rather than imposing any particular parametric input-output relationship in advance (e.g., linear, polynomial), Gaussian process models extract this information directly from the data. The package offers built-in dimension reduction, which helps to simplify the representation of the functional inputs and obtain lighter models. It also implements an Ant Colony based optimization algorithm which supports the calibration of multiple structural characteristics of the model such as the state of each input (active or inactive) and the type of kernel function, while seeking for greater prediction power. The implemented methods are tested and applied to a real case in the domain of marine flooding. The **funGp** package is downloadable from GitHub (https://github.com/djbetancourt-gh/funGp) and CRAN (https://cran.r-project.org/package=funGp).

Keywords: Gaussian process, metamodeling, functional inputs, computer experiments, R.

What does funGp bring to the table?

- **Flexible modeling of functional-input regression problems**
  A narrow class of R packages address regression with functional inputs (e.g., time series). The vast majority of those packages rely on models limited by strong assump- tions on the relationship between inputs and outputs (e.g., Linear, Generalized Linear or Generalized Additive Models). The few ones that suppress these limitations through more general models (e.g., Kernel Smoothing) often require the output to be a function defined over the same domain as the functional inputs, which is frequently not the case and leaves the scalar-output problem unresolved. **funGp** tackles regression problems involving scalar and/or functional inputs and a scalar output through the fairly general Gaussian process model. This is a non-parametric type of model which removes any need to set a particular input-output parametric relationship in advance, and learns this information directly from the data.

- **Built-in dimension reduction**
  A common practice when working with functional data is to start by making a projection of it onto a space of lower dimension, a procedure known as dimension reduction (DR). This allows to reduce the complexity of the model while preserving the main statistical or geometric characteristics of the functions. **funGp** is self-contained in the sense that it does not depend on other packages to perform DR on the functional inputs. At this point, we provide projection onto B-splines or PCA bases. The package was designed to enable a straightforward extension to other bases in further versions.

- **Heuristic model selection**
  The possibilities offered by a package often translate into alternative model structures. Just to give an example, most packages that support Gaussian process models allow to select the kernel function from a set of standard families (e.g., Gaussian, Matérn 5/2, Matérn 3/2). However, decision support is rarely offered in order to select a suitable configuration for the problem at hand. We acknowledge the potential impact of such a decision in the performance of the model Betancourt, Bachoc, Klein, Idier, Pedreros, and Rohmer (2020b); Lataniotis, Marelli, and Sudret (2020) and also the practical difficulties that arise from offering possibilities without decision support. Thus, **funGp** was equipped with a model selection functionality that allows the user to automatically search for a good combination of the so-called structural parameters of the model. At this point, an Ant-Colony-based algorithm is implemented to perform this task.

- **All-level-user-friendly**
  We aim **funGp** to be a helpful tool for users within a wide range of knowledge in mathematics or statistics. Thus, we have made an effort to make simple and intuitive the way the package work. Most of the arguments in the functions have been provided default values so that the user can start experimenting with them at its own pace. Once you get ready, you will be able to start playing with the nugget effect, basis type, kernel type, multi-start option, parallelization and even the parameters of the heuristic for model selection. However, to have your first model built by **funGp**, the only thing you need to provide is your data.

1. Introduction

Gaussian process (GP) models (Sacks, Welch, Mitchell, and Wynn 1989; Oakley and O'Hagan 2002) are one of the most popular regression methods thanks to the great flexibility they offer in the representation of complex non-linear input-output relationships, their high prediction power, interpretability, and ability to provide both an interpolation of the data and an uncertainty quantification in the unexplored regions, e.g., Marrel, Iooss, Van Dorpe, and Volkova (2008). The GP model was first developed in spatial statistics under the name of *kriging* (Cressie 1990) and rapidly gained attention in the machine learning community where it was classified as a *kernel machine* (Rasmussen and Williams 2006). Over the years, the GP model has become popular in a wide range of applications such as nuclear safety (Bachoc, Ammar, and Martinez 2016), wind power generation (Mori and Kurata 2008), vehicle design and navigation (Chen, Dai, Wang, and Liu 2014), modeling and prediction of natural hazards (Rohmer and Idier 2012; Liu and Guillas 2017), among several others.

Here, a special focus will be given to the use of GP models as metamodeling techniques in the field of *design and analysis of computer experiments*, where the regression models are used as surrogates of expensive-to-run numerical models (see e.g., Yuan and Nian (2018) and Iooss and Marrel (2019)). Indeed, in recent years, the use of computer codes for the study of complex systems has become a widespread practice to simulate systems at a lower resource expense owing to lack of observations, in particular regarding the input-output conditions for some natural processes (e.g., volcanic activity, earthquakes, flooding, etc.). Despite the increasing computer power over the last years, computer codes for environmental and industrial applications are often too time-consuming for direct application (e.g., for uncertainty quantification or fast prediction within an early warning system). A possible option to overcome this computational burden is to set up quick-to-evaluate mathematical emulators of those numerical codes (Forrester, Sobester, and Keane 2008; Simpson, Poplinski, Koch, and Allen 2001), based on a limited collection of model runs within the generic framework of design of computer experiments (Santner, Williams, Notz, and Williams 2003); such emulators being often called surrogate models or metamodels.

As shown in Østergård, Jensen, and Maagaard (2018) through comparison in 13 different metamodeling instances, GP metamodels are often able to outperform other popular methods, such as *neural networks* (Fonseca, Navaresse, and Moynihan 2003), *multivariate adaptive regression splines* (Friedman 1991) and *support vector regression* (Drucker, Burges, Kaufman, Smola, and Vapnik 1997), in terms of prediction accuracy.

In regression, a large amount of research works have addressed the case of scalar-valued inputs and outputs, e.g., (Forrester *et al.* 2008; Simpson *et al.* 2001; Santner *et al.* 2003). However, industrial and environmental applications often deal with complex inputs or outputs, which can be spatial, temporal, spatio-temporal i.e., functional in a more general way. As a motivating real case example, the coastal flood early warning system (FEWS) developed at Gâvres, France (Idier, Aurouet, Bachoc, Bailly, Betancourt, Gamboa, Klein, López-Lopera, Pedreros, Rohmer, and Thibault 2021) predicts scalar output indicators of flooding (like total flooded area, maximum volume of water entering the territory, maximum water depth at a given location, etc.), based on inputs which are multiple scalar and time-varying maritime conditions (e.g., mean sea level, tide, atmospheric storm surge, and wave conditions), i.e., multiple time series (that are here sampled regularly over time). This FEWS is based on an hydrodynamic numerical model, but, to reach computation times compatible with short term

forecast, metamodels had to be developed taking into account these inputs and providing the listed outputs.

Although GPs for scalar-valued inputs and outputs have been studied for almost 30 years, solving the problem of metamodeling in the functional framework (as described for the FEWS example) is still a much less developed research area (Muehlenstaedt, Fruth, and Roustant 2017). In principle one could use any GP implementation to fit a model with functional inputs by representing each function as a vector, and then using each element of it as an individual scalar input of the model. However, this might lead to challenging and long hyperparameter optimization processes due to the large number of resulting decision variables. Furthermore, this approach disregards the functional nature of the inputs. To fill these gaps, this paper presents the package **funGp** in R programming environment (R Core Team 2020). It provides the construction of GP regression models of a single scalar output, dealing at the same time with multiple functional and scalar inputs by restricting on the situations where the functional inputs are regularly sampled time series (time series are considered for convenience of exposition; the **funGp** package can apply to other types of one-dimensional inputs as well). The **funGp** package offers dimension reduction (DR) methods which allow the projection of functional inputs onto a space of lower dimension while preserving the main statistical or geometric properties of each variable (Ramsay and Silverman 2007; Nanty, Helbert, Marrel, Pérot, and Prieur 2016). Then, **funGp** provides the standard functionalities of GP models such as prediction, computation of confidence intervals, and conditional simulation. The second main contribution of **funGp** is an automatic procedure for the selection of the optimal *structural parameters*: (i) the state of each input (active or inactive), (ii) the DR method (e.g., PCA, B-splines), (iii) projection dimension and (iv) distance function used for each functional input, and (v) the type of kernel for the model (e.g., Gaussian, Matérn 5/2).

Table 1 provides a comparison between the functionalities of 21 classical open-source packages that either implement GP regression or support scalar-on-function regression with the GP or other types of models. This list includes the benchmark packages listed in the state of the art in functional data analysis in R presented by Febrero-Bande and Oviedo de la Fuente (2012) and later updated in the website `https://rpubs.com/moviedo/fda_usc_introduction`, and also in the state of the art in software for GP regression presented in Roustant, Ginsbourger, and Deville (2012). To our best knowledge, and as we can see from our classification, **funGp** is the first open-source software that offers GP regression with scalar and functional inputs together with an automatic optimization method to set up the structural parameters of the model.

The remainder of this paper is structured as follows. In Section 2 we provide the necessary statistical background. Then, in Section 3 we explain the four fundamental functionalities of **funGp**: creation and diagnostic of GP regression models (Section 3.1), prediction (Section 3.2), simulation (Section 3.3) and model updating (Section 3.4). In Section 4 we show how the structural parameters of a `funGp` model can be set up manually. Then, in Section 5 we explain how to delegate the automatic structural configuration of the model to **funGp**. In Section 6 we show how the procedures of model construction and structural configuration can be sped up by means of parallelized implementations available in **funGp**. Section 7 applies **funGp** functionalities to a real case in the domain of marine flooding. Finally, Section 8 concludes. A compact overview of the package including a list of its main programming functions is provided in Appendix A.

Software/tool	Ref.	Platform	Model		Functional inputs	Structural configuration	
			GP	Other		Automatic	Assisted
GPML	[1]	MATLAB	✓				
GPstuff	[2]	MATLAB	✓				
GPy	[3]	Python	✓				
Scikit-learn	[4]	Python	✓	✓		✓[1]	
GPflow	[5]	Python	✓				
NPFDA	[6]	R		✓	✓	✓[2]	
splinetree	[7]	R		✓			
RRegrs	[8]	R		✓			✓[3]
fda	[9]	R		✓	✓		
fdaMixed	[10]	R		✓			
fda.usc	[11]	R		✓	✓		✓[4]
fdapace	[12]	R		✓	✓	✓[5]	
goffda	[13]	R		✓	✓		
refund	[14]	R		✓	✓	✓[5]	
FDboost	[15]	R		✓	✓		
flars	[16]	R		✓	✓	✓[4]	
growfunctions	[17]	R	✓	✓			
RobustGaSP	[18]	R	✓			✓[4]	
DiceKriging	[19]	R	✓				
GPFDA	[20]	R	✓		✓		
kergp	[21]	R	✓		✓*		
funGp	[22]	R	✓		✓	✓	

Table 1: Main open-source statistical tools either implementing the GP model or providing functional-input regression. Convention: (✓*) the user is allowed to pass a custom kernel, but no explicit treatment of functional inputs is offered; (✓[1]) automatic selection between various types of models (e.g., GP, neural networks), but no optimization of structural parameters; (✓[2]) automatic bandwidth selection for a kernel k nearest neighbors model; (✓[3]) performance statistics for several types of models are provided to help the user select one; (✓[4]) support for variable selection; (✓[5]) automatic selection of the number of principal components in a PCA projection. [1] Rasmussen and Nickisch (2010), [2] Vanhatalo *et al.* (2012), [3] GPy (2012), [4] Pedregosa *et al.* (2011), [5] De G. Matthews *et al.* (2017), [6] Ferraty and Vieu (2006), [7] Neufeld and Heggeseth (2019), [8] Tsiliki *et al.* (2015), [9] Ramsay *et al.* (2022), [10] Markussen (2019), [11] Febrero-Bande and Oviedo de la Fuente (2012), [12] Gajardo *et al.* (2021), [13] García-Portugués and Álvarez Liébana (2021), [14] Goldsmith *et al.* (2020), [15] Brockhaus and Ruegamer (2018), [16] Cheng and Shi (2016), [17] Savitsky (2016), [18] Gu *et al.* (2020), [19] Roustant *et al.* (2012), [20] Shi and Cheng (2014), [21] Deville *et al.* (2020), [22] Betancourt *et al.* (2020c).

2. Statistical background

2.1. Structure of the regression problem

The **funGp** package considers the approximation of a real-valued function $\mathcal{F}_{\text{sys}}$ by means of a GP regression model. Throughout the article, we discriminate between three possible settings regarding the inputs of $\mathcal{F}_{\text{sys}}$:

(a) When the function is $\boldsymbol{x} \mapsto \mathcal{F}_{\text{sys}}(\boldsymbol{x})$, with $\boldsymbol{x} = \left(x^{(1)}, \ldots, x^{(d_s)}\right)^\top$ and $x^{(k)} \in \mathbb{R}$ for

$k = 1, \ldots, d_s$, we say that the system has d_s scalar inputs, we call $x^{(k)}$ for $k = 1, \ldots, d_s$ a scalar input, and we call $\boldsymbol{x}$ a vector of scalar inputs. For simplicity, we may also refer to $\boldsymbol{x}$ as scalar inputs.

(b) When the function is $\boldsymbol{f} \mapsto \mathcal{F}_{\text{sys}}(\boldsymbol{f})$, with $\boldsymbol{f} = \left(f^{(1)}, \ldots, f^{(d_f)}\right)^\top$ and $f^{(k)} : \mathcal{T}_k \subset \mathbb{R} \to \mathbb{R}$ for $k = 1, \ldots, d_f$, we say that the system has d_f functional inputs, we call $f^{(k)}$ for $k = 1, \ldots, d_f$ a functional input, and we call $\boldsymbol{f}$ a vector of functional inputs. For simplicity, we may also refer to $\boldsymbol{f}$ as functional inputs. We also denote by $\boldsymbol{\mathcal{F}}$ the set of all $\left(f^{(1)}, \ldots, f^{(d_f)}\right)^\top$ with $f^{(k)} : \mathcal{T}_k \subset \mathbb{R} \to \mathbb{R}$ for $k = 1, \ldots, d_f$.

(c) When the function is $(\boldsymbol{x}, \boldsymbol{f}) \mapsto \mathcal{F}_{\text{sys}}(\boldsymbol{x}, \boldsymbol{f})$, we say that the system has d_s scalar inputs and d_f functional inputs, and we use the same vocabulary as before for $\boldsymbol{x}$ and $\boldsymbol{f}$. In addition, we may refer to this setting as the hybrid-input case.

2.2. Gaussian process regression

Let us consider in the rest of Section 2 the hybrid-input case, of which the scalar- and functional-input ones are special cases. The goal is to build an approximate model for $\mathcal{F}_{\text{sys}}$ using a learning set $D = \{(\boldsymbol{x}_1, \boldsymbol{f}_1, \mathcal{F}_{\text{sys}}(\boldsymbol{x}_1, \boldsymbol{f}_1)), \ldots, (\boldsymbol{x}_n, \boldsymbol{f}_n, \mathcal{F}_{\text{sys}}(\boldsymbol{x}_n, \boldsymbol{f}_n))\}$. The GP model treats the fixed function $\mathcal{F}_{\text{sys}}$ as a realization of a Gaussian (random) process ξ, specified by its mean and covariance functions μ and C. Here ξ is assumed to be centered ($\mu = 0$) and it is thus specified by its covariance function C:

$$\xi(\cdot) \sim \mathcal{GP}(0, C(\cdot, \cdot)), \tag{1}$$

where the kernel $C((\boldsymbol{x}, \boldsymbol{f}), (\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{f}})) = \mathsf{COV}(\xi(\boldsymbol{x}, \boldsymbol{f}), \xi(\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{f}}))$ evaluates the covariance for any pair of input vectors $(\boldsymbol{x}, \boldsymbol{f}), (\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{f}}) \in \mathbb{R}^{d_s} \times \boldsymbol{\mathcal{F}}$. For instance, if $(\boldsymbol{x}, \boldsymbol{f}) = (\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{f}})$, then $C((\boldsymbol{x}, \boldsymbol{f}), (\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{f}}))$ is large and corresponds to perfect correlation. Otherwise, the covariance tends to zero as the distance between the two inputs increases.

One of the main benefits of GP models lies in the tractability of conditional distributions, which constitute the core mechanism to perform prediction and simulation. In the GP model, predicting the value of $\mathcal{F}_{\text{sys}}$ at a set of unobserved input points amounts to conditioning ξ on the learning set D. Conditional on $\xi(\boldsymbol{x}_1, \boldsymbol{f}_1) = \mathcal{F}_{\text{sys}}(\boldsymbol{x}_1, \boldsymbol{f}_1), \ldots, \xi(\boldsymbol{x}_n, \boldsymbol{f}_n) = \mathcal{F}_{\text{sys}}(\boldsymbol{x}_n, \boldsymbol{f}_n)$, ξ is a GP with conditional mean values at $(\boldsymbol{X}_*, \boldsymbol{F}_*) = (\boldsymbol{x}_{*1}, \ldots, \boldsymbol{x}_{*n_*}, \boldsymbol{f}_{*1}, \ldots, \boldsymbol{f}_{*n_*})$ given by the vector

$$C((\boldsymbol{X}_*, \boldsymbol{F}_*), (\boldsymbol{X}, \boldsymbol{F}))\; C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}, \boldsymbol{F}))^{-1}\; \boldsymbol{y} \tag{2}$$

and conditional covariance matrix at the same n_* points given by

$$C((\boldsymbol{X}_*, \boldsymbol{F}_*), (\boldsymbol{X}_*, \boldsymbol{F}_*)) - C((\boldsymbol{X}_*, \boldsymbol{F}_*), (\boldsymbol{X}, \boldsymbol{F}))\; C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}, \boldsymbol{F}))^{-1}\; C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}_*, \boldsymbol{F}_*)). \tag{3}$$

In the expressions above, $\boldsymbol{y} = (\mathcal{F}_{\text{sys}}(\boldsymbol{x}_1, \boldsymbol{f}_1), \ldots, \mathcal{F}_{\text{sys}}(\boldsymbol{x}_n, \boldsymbol{f}_n))^\top$, $C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}_*, \boldsymbol{F}_*))$ is the $n \times n_*$ unconditional covariance matrix between the observed values and the values to be predicted, $C((\boldsymbol{X}_*, \boldsymbol{F}_*), (\boldsymbol{X}, \boldsymbol{F})) = C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}_*, \boldsymbol{F}_*))^\top$, $C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}, \boldsymbol{F}))$ is the $n \times n$ unconditional covariance matrix of the observed values and $C((\boldsymbol{X}_*, \boldsymbol{F}_*), (\boldsymbol{X}_*, \boldsymbol{F}_*))$ is the $n_* \times n_*$ unconditional covariance matrix of the values to be predicted. These formulas can be found for instance in Rasmussen and Williams (2006).

The covariance function C is determined by the structural parameters listed in Section 1 (the state of each scalar and functional input, the DR method, projection dimension and distance function used for each functional input, and the type of kernel function for the model). For a given choice of these structural parameters, C is decomposed as a variance σ^2 times the tensor product of a correlation function for the scalar inputs and a correlation function for the functional inputs:

$$C((\boldsymbol{x}, \boldsymbol{f}), (\tilde{\boldsymbol{x}}, \tilde{\boldsymbol{f}})) = \sigma^2 \; R(\boldsymbol{x} - \tilde{\boldsymbol{x}}; \boldsymbol{\theta}_s) \; R\Big(\boldsymbol{f} - \tilde{\boldsymbol{f}}; \boldsymbol{\theta}_f\Big),$$

where $\boldsymbol{\theta}_s$ and $\boldsymbol{\theta}_f$ are the correlation parameters. For any choice of the structural parameters, the covariance parameters σ^2, $\boldsymbol{\theta}_s$ and $\boldsymbol{\theta}_f$ (a.k.a. hyperparameters) are selected by optimization of the likelihood computed on the learning set. We provide further details on the precise form of the covariance function implemented in **funGp** later in Section 4. Complementary information on our modeling approach can also be found in Betancourt *et al.* (2020b).

2.3. Optimization of the structural parameters

As explained above, a core feature of **funGp** is the possibility to optimize the structural parameters of the regression model to achieve better predictive performance. So far, for GP models, most research has exclusively focused on variable selection, meaning the optimization of the state of the input variables (Marrel *et al.* 2008; Lee and Park 2017; Ben Salem, Bachoc, Roustant, Gamboa, and Tomaso 2019). This task is typically performed through *sensitivity analysis*, which seeks to apportion the variation in the output variable to variations in the input variables (Saltelli, Tarantola, Campolongo, and Ratto 2004). By contrast, **funGp** treats the structural configuration of the model as a combinatorial optimization task, where each decision variable is linked to one structural parameter and the predictive power of the model is used as the objective function. By now, **funGp** offers structural optimization based on the well known Ant Colony Optimization (ACO) algorithm (Dorigo and Gambardella 1997), which has been recognized as one of the most successful research lines in the area of swarm intelligence (Bonabeau, Dorigo, and Theraulaz 1999) and is always listed among the preferred metaheuristic optimization techniques (Blum 2005). We validated the ability of our ACO-based algorithm to find high performing model configurations in more than a dozen instances. The results obtained in various of those tests can be found in Chapters 3-5 of Betancourt (2020). We describe the functioning of this algorithm more in depth in Section 5.

3. Base funGp functionalities

In this section, we start by explaining the four fundamental procedures available in **funGp**: (i) creation and diagnostic of regression models, (ii) prediction of the output at unobserved input points, (iii) simulation of trajectories from the underlying GP linked to any `funGp` model, and (iv) update of an existing model. The workflow for each of these four functionalities is illustrated through a follow-along example which considers the following analytic black-box function to regress:

$$\begin{aligned} \mathcal{G}: \quad & [0,1]^2 \times \mathcal{F} \to \mathbb{R}, \\ & (\boldsymbol{x}, \boldsymbol{f}) \mapsto x^{(1)} + 2x^{(2)} + 4 \int_0^1 t f^{(1)}(t) \; dt + \int_0^1 f^{(2)}(t) \; dt, \end{aligned}$$

with $\boldsymbol{x} = \left(x^{(1)}, x^{(2)}\right)^\top$ the scalar inputs, $\boldsymbol{f} = \left(f^{(1)}, f^{(2)}\right)^\top$ the functional inputs, and $\mathcal{F}$ the set of pairs of continuous functions from $[0,1]$ to $\mathbb{R}$. This function corresponds to the first analytic example in Muehlenstaedt *et al.* (2017), and is accessible in **funGp** through `fgp_BB3()`. By construction, function $\mathcal{G}$ has the first scalar and second functional input of the same importance, whereas the second scalar input and the first functional input are comparably influential but are more important than the first scalar and the second functional input.

3.1. Create a funGp model

Let us start by creating a model. To do so, we must first put the input and output data in a suitable format. The scalar inputs should be provided as a `matrix` or `data.frame`. The functional inputs should be provided as a `list` of `matrices`, one per functional input. When the data set contains inputs of the form $\boldsymbol{x}_1, \boldsymbol{f}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{f}_n$ (see Section 2), the scalar input matrix is $n \times d_s$, with line ℓ containing the vector $\boldsymbol{x}_\ell = (\boldsymbol{x}_\ell^{(1)}, \ldots, \boldsymbol{x}_\ell^{(d_s)})$. The functional input list is of size d_f. In this list, the matrix corresponding to the j-th functional input has size $n \times N^{(j)}$ and its line ℓ contains the vector $(f_\ell^{(j)}(t_1^{(j)}), \ldots, f_\ell^{(j)}(t_{N^{(j)}}^{(j)}))$, where $t_1^{(j)}, \ldots, t_{N^{(j)}}^{(j)} \in \mathcal{T}_j$ are the time instants at which the input function $f_\ell^{(j)}$ is measured. Note that the input functions can thus also be thought of as time series inputs. Note also that the time instants should be equispaced, as integrals (see Section 4.4) related to the j-th functional input are numerically implemented as averages over the $N^{(j)}$ time instants.

The output should be provided as an `array` or single-column `matrix`. Here, we will use synthetic data based on the analytic case defined above, which involves two scalar inputs and two functional inputs. To generate the scalar input data, we set up a factorial design over $[0,1]^2$. For the functional inputs, we assume that $f^{(1)}$ is measured at 10 time instants, $f^{(2)}$ at 22, and then we sample all the values of each function randomly from the standard uniform $U(0,1)$. We use an arbitrary number of 25 training points for this example.

```
R> library("funGp")
R> ## generating input and output data for training
R> set.seed(100)
R> n.tr <- 25
R> sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)),
+                     x2 = seq(0,1,length = sqrt(n.tr)))
R> fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10),
+              f2 = matrix(runif(n.tr*22), ncol = 22))
R> sOut <- fgp_BB3(sIn, fIn, n.tr)
R> ## creating a funGp model
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut)

** Presampling...

** Optimising hyperparameters...

final  value 2.841058
converged
```

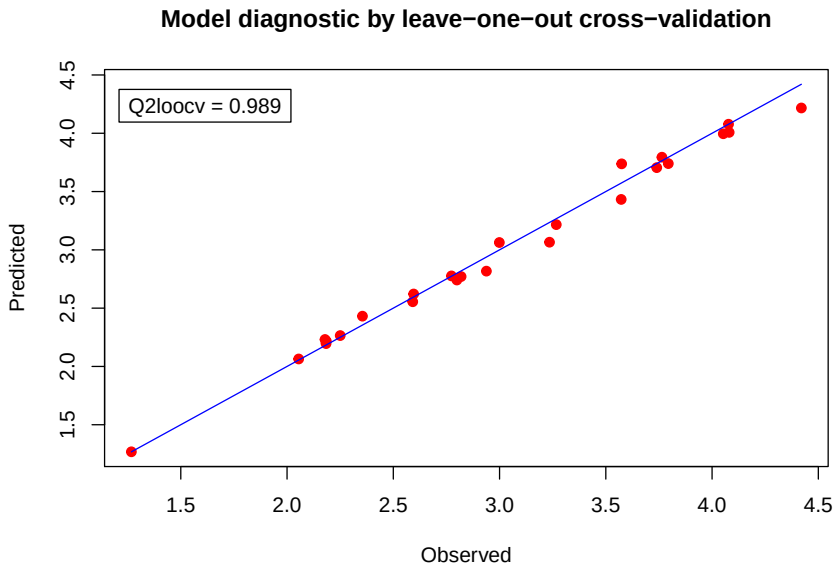


Figure 1: Model diagnostic by leave-one-out cross-validation.

```
** Hyperparameters done!
```

It should be noted that, above, the function `fgpm()` does not take any input of type `formula`, although it provides a fitted regression model. This is in contrast, for instance, with the creation of linear models with the function `lm()` in R. The reason for this difference is that linear models are parametric and require the user to pre-specify regression functions that are linearly combined for prediction, for instance monomial functions of the inputs. In contrast, centered GPs provide a non-parametric regression model, where the predictions are fully data-driven and not restricted to combinations of user-specified functions. Note that if non-centered GPs were considered, it would be possible to select the mean functions as linear combinations of pre-specified functions, which could be done with a `formula` input to the function `fgpm()`. This would correspond to using universal Kriging equations instead of simple Kriging ones (see for instance Roustant *et al.* 2012) in Section 2. In this case, the predictions would still be non-parametric, but with a parametric component. This may be a topic of future extension of the **funGp** package.

The output of `fgpm()` is an object of class `"fgpm"` representing the fitted model. The next examples in Section 3 will reuse this model to illustrate other functionalities of the package. A first diagnostic procedure of the model can be made by calling

```
R> plot(m1)
```

which will display a calibration plot based on the leave-one-out (LOO) predictions, as shown in Figure 1. For a design with n_{tr} points, LOO consists in building the model n_{tr} times, each using a different set of $n_{\text{tr}} - 1$ points for training and computing the prediction at the ignored point. The plot will also display the LOO cross-validated squared correlation coefficient Q^2_{loocv}

as a measure of the external prediction capability of the model:

$$Q^2_{\text{loocv}} := 1 - \frac{\sum_{i=1}^{n_{\text{tr}}} (y_i - \hat{y}_{i,-i})^2}{\sum_{i=1}^{n_{\text{tr}}} (y_i - \bar{y})^2},$$

with $(y_i)_{i=1,\dots,n_{\text{tr}}}$ the vector of observed output values ($y_i = \mathcal{F}_{\text{sys}}(\boldsymbol{x}_i, \boldsymbol{f}_i)$ in the hybrid-input case), $\bar{y}$ the average of that vector and $\hat{y}_{i,-i}$ the LOO prediction of y_i. Since the classical LOO procedure might be time-expensive, **funGp** implements the virtual LOO formulas (Dubrule 1983) that require a single model training.

The main features of the model are printed by means of the `summary()` method:

```
R> summary(m1)
```

```
Gaussian Process Model_____________________________________
* Scalar inputs: 2
* Functional inputs: 2

 Input    Orig. dim    Proj. dim      Basis       Distance
-------  -----------  -----------  -----------  ------------
  F1         10            3        B-splines    L2_bygroup
  F2         22            3        B-splines    L2_bygroup

* Total data points: 25
* Trained with: 25
* Kernel type: matern5_2
* Hyperparameters:
  -> variance: 1.6404
  -> length-scale:
         ls(X1): 2.0000
         ls(X2): 2.0000
         ls(F1): 2.5804
         ls(F2): 3.0370
------------------------------------------------------------
```

The field `Proj. dim` relates to the possibility of performing DR (dimension reduction) on the functional inputs. Recall from the introduction that DR allows the projection of a functional input of dimension q onto a space of lower dimension p while preserving the main statistical or geometric properties of the variable. This process typically leads to $p \ll q$, which improves the tractability and processing speed of the model. By default, the `fgpm()` function sets $p = 3$ for every functional input. However, the user is allowed to pick custom projection dimensions and also not to project some inputs. The projection method used for each input is indicated under the field `Basis`; at this point, the implemented options are B-splines and PCA. Other methods might be integrated in the near future. The details on the choice of the projection method and dimension are discussed in Sections 4.2 and 4.3, respectively.

3.2. Predict using a `funGp` model

Now let us use our model to make predictions. To do so, we must prepare the input data corresponding to the coordinates at which the output is to be estimated. The inputs should have the same format as used for creating the model with the `fgpm()` function in Section 3.1. The scalar inputs should be provided as a `matrix` or `data.frame` and the functional inputs should be provided as a `list` of `matrices`, one per functional input. This time, each row of an input `matrix` must correspond to a prediction point. For the example, we generate prediction points in a similar way as the training points, i.e., the scalar inputs from a factorial design over $[0, 1]^2$ and the functional values randomly sampled from the $U(0, 1)$ distribution.

```
R> ## generating input data for prediction
R> n.pr <- 100
R> sIn.pr <- as.matrix(expand.grid(x1 = seq(0, 1, length = sqrt(n.pr)),
+                                  x2 = seq(0, 1, length = sqrt(n.pr))))
R> fIn.pr <- list(f1 = matrix(runif(n.pr * 10), ncol = 10),
+                 f2 = matrix(runif(n.pr * 22), ncol = 22))
```

Predictions can be requested by passing a `funGp` model and the prediction points to the `predict()` function. For instance, we can use the model `m1` defined in Section 3.1.

```
R> m1.preds <- predict(m1, sIn.pr = sIn.pr, fIn.pr = fIn.pr)
R> t(data.frame(row.names = names(m1.preds), Length = lengths(m1.preds)))
```

```
       mean  sd lower95 upper95
Length  100 100     100     100
```

The output of `predict()` is mainly a `list` containing the estimated mean, standard deviation, and limits of the 95% confidence intervals for the output at the prediction points. In practice, the estimated mean of a GP model is used as the prediction of the output, while the standard deviation is often interpreted as a measure of the local error of the prediction. Actually the object returned by the `predict()` method on a `funGp` object has S3 class `"predict.fgpm"` inheriting from `"list"`. This allows to use the `plot()` method for this S3 class.

```
R> plot(m1.preds)
```

The resulting plot shows the increasingly sorted mean and corresponding confidence intervals as displayed in Figure 2. This plot can be used as a diagnostic tool for the hyperparameter optimization process. In this regard, overly wide confidence intervals for a large proportion of prediction points or missing confidence intervals for points out of the training set, as shown in Figure 3, might indicate a far-from-optimal solution.

The `plot()` method can also be used to compare predictions against true output values. For that aim, the vector of "true" output values at the prediction points should be passed in the optional argument `sOut.pr`, along with the object returned by `predict()`.

```
R> sOut.pr <- fgp_BB3(sIn.pr, fIn.pr, n.pr)
R> plot(m1.preds, sOut.pr = sOut.pr)
```

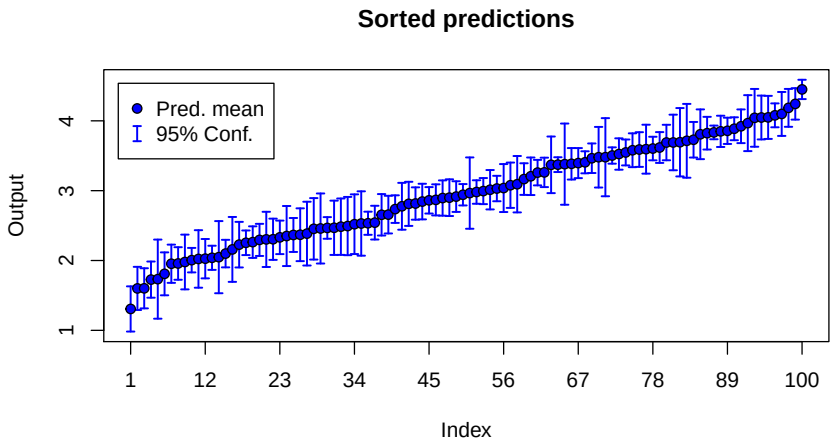


Figure 2: Predictions of a `funGp` model sorted in increasing order.

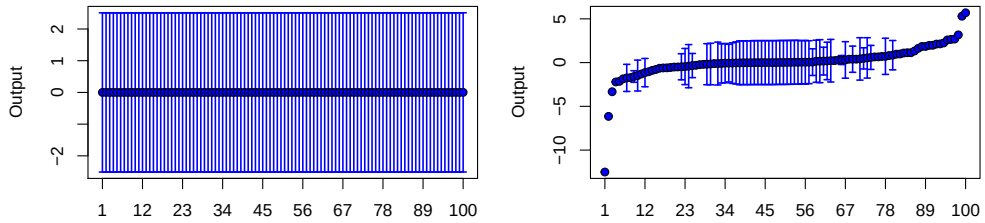


Figure 3: Signs of difficulties during the hyperparameter optimization. Left: the confidence intervals are too wide. Right: some confidence intervals are too thin.

In this case, a calibration plot will be placed on top of a sorted-output plot as shown in Figure 4. The calibration plot will display the predictive squared correlation coefficient Q^2_{hout}, corresponding to the classical coefficient of determination R^2 for a test sample (Nilsson, de Jong, and Smilde 1997). The ordering in the sorted-output plot will be lead by the true output vector to facilitate the comparison of different models fitting the same data.

By default the `predict()` function in **funGp** returns so-called *light predictions*, which include the predicted mean, standard deviation and limits of the 95% confidence intervals. Some users might be interested in *full predictions*, which also include the unconditional training-prediction cross-covariance matrix `K.tp` and the unconditional prediction auto-covariance matrix `K.pp`, which can be used for verification purposes (see Appendix B). To make full predictions, it suffices to set `detail = "full"` when calling `predict()`, as shown below.

```
R> m1.preds <- predict(m1, sIn.pr = sIn.pr, fIn.pr = fIn.pr, detail = "full")
R> t(data.frame(row.names = names(m1.preds), Length = lengths(m1.preds)))
```

```
       mean  sd K.tp  K.pp lower95 upper95
Length  100 100 2500 10000     100     100
```

3.3. Simulate from a `funGp` model

Conditional simulations in **funGp** are requested through `simulate()`, similarly as predictions are requested through `predict()`. The scalar inputs should be provided as a `matrix` or

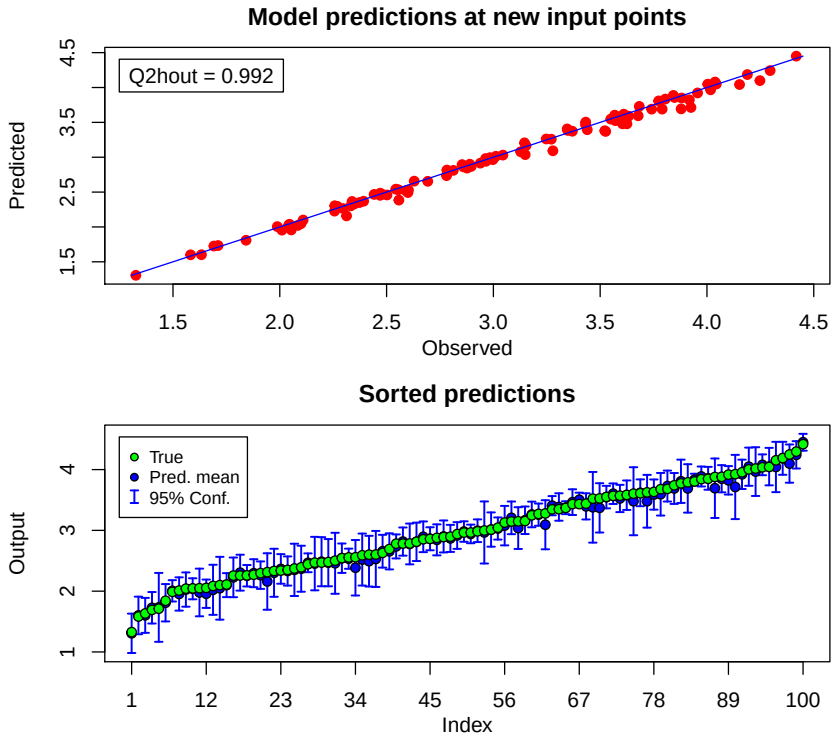


Figure 4: Prediction plot using the `plot()` method on the object returned by `predict()`.

`data.frame` and the functional inputs should be provided as a `list` of `matrices`, one per functional input. Each row of an input `matrix` is interpreted as a point at which to provide simulations. By default, `simulate()` will perform so-called *light simulations*, thus returning a $n.rep \times n.sm$ `matrix`, with $n.rep$ replications at $n.sm$ input points. For this example we use the model `m1` created in Section 3.1. In addition, we take the scalar input values from a factorial design over $[0, 1]$ and the functional values randomly from $U(0, 1)$.

```
R> ## generating input points for simulation
R> n.sm <- 100
R> sIn.sm <- as.matrix(expand.grid(x1 = seq(0, 1, length = sqrt(n.sm)),
+                                  x2 = seq(0, 1, length = sqrt(n.sm))))
R> fIn.sm <- list(f1 = matrix(runif(n.sm * 10), ncol = 10),
+                 f2 = matrix(runif(n.sm * 22), ncol = 22))
R> ## making light simulations
R> m1.sims_l <- simulate(m1, nsim = 10, sIn.sm = sIn.sm, fIn.sm = fIn.sm)
```

Simulations in **funGp** are plotted by the `plot()` method for the S3 class `"simulate.fgpm"`. In contrast to prediction plots, simulation plots do not have the output sorted in increasing order, but instead in the order given by the simulation coordinates specified by the user. This plot is illustrated in Figure 5.

```
R> plot(m1.sims_l)
```

If requested, `simulate()` will return a `list` containing the simulated output, predicted mean,

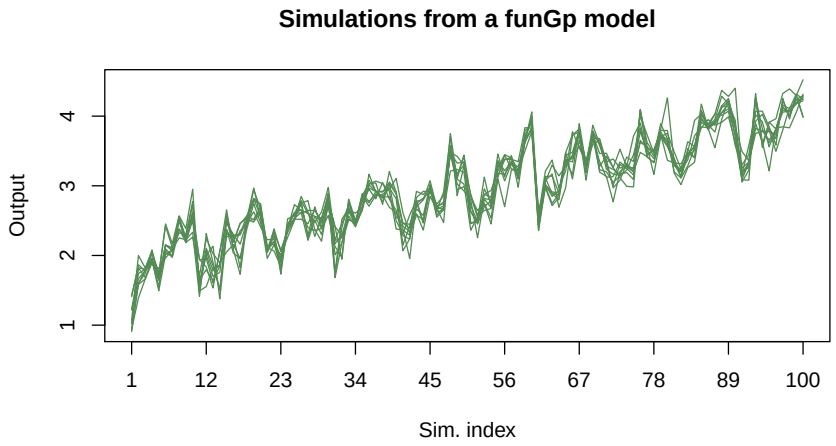


Figure 5: Simulations from a `funGp` model.

standard deviation and limits of the 95% confidence intervals at the specified input coordinates. This corresponds to a *full simulation*, available through the option `detail = "full"`.

```
R> m1.sims_f <- simulate(m1, nsim = 10, sIn.sm = sIn.sm, fIn.sm = fIn.sm,
+                        detail = "full")
R> t(data.frame(row.names = names(m1.sims_f), Length = lengths(m1.sims_f)))

       sims mean  sd lower95 upper95
Length 1000  100 100     100     100
```

Full simulations can be plotted similarly as light ones by means of the `plot()` function.

```
R> plot(m1.sims_f)
```

By default, the full simulation plot will include the predicted mean and limits of the confidence intervals (Figure 6). A light plot with just the simulated output values can also be requested for full simulations by setting the argument `detail = "light"` in the `plot()` call.

3.4. Update of a `funGp` model

Models created by **funGp** can be easily modified at any time by means of the `update()` function, which incorporates the following nine updating operations:

1. Deletion of data points;
2. Substitution of data points;
3. Addition of data points;
4. Substitution of the variance hyperparameter;
5. Substitution of the vector of scalar length-scale hyperparameters;
6. Substitution of the vector of functional length-scale hyperparameters;
7. Re-estimation of the variance hyperparameter;
8. Re-estimation of the vector of scalar length-scale hyperparameters;

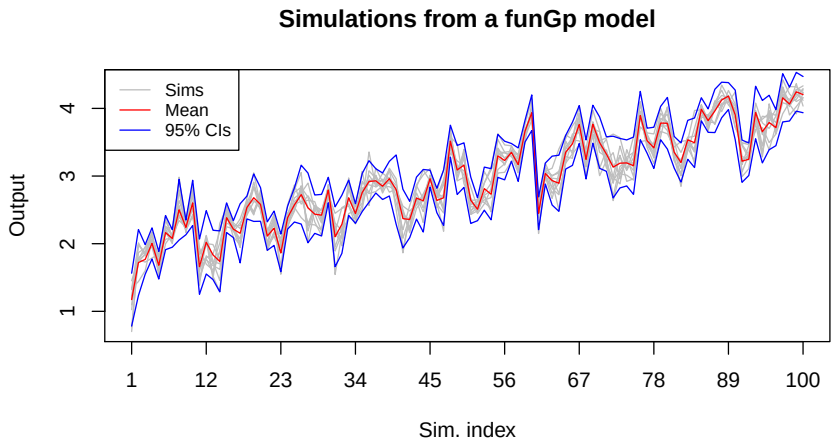


Figure 6: Simulations from a `funGp` model with confidence intervals.

9. Re-estimation of the vector of functional length-scale hyperparameters.

Through `update()`, **funGp** exploits the work done for the construction of the original model, making the updating process much faster than building a new model from zero. The request of updates is illustrated below through examples based on the model `m1` created in Section 3.1.

◇ ***Deletion and addition of data points***

```
R> ## deleting two points
R> ind.dl <- sample(1:m1@n.tot, 2)
R> m1up <- update(m1, ind.dl = ind.dl)

--------------
Update summary
--------------
* Complete tasks:
  - data deletion

R> ## adding five points
R> n.nw <- 5
R> sIn.nw <- matrix(runif(n.nw * m1@ds), nrow = n.nw)
R> fIn.nw <- list(f1 = matrix(runif(n.nw * 10), ncol = 10),
+                 f2 = matrix(runif(n.nw * 22), ncol = 22))
R> sOut.nw <- fgp_BB3(sIn.nw, fIn.nw, n.nw)
R> m1up <- update(m1, sIn.nw = sIn.nw, fIn.nw = fIn.nw, sOut.nw = sOut.nw)

--------------
Update summary
--------------
* Complete tasks:
  - data addition
```

A more extensive study shows us that the computation time cost is largely decreased when using the `update()` function. With 25 additional points, it is reduced by 50%, with 100, it is reduced by nearly 80%.

◇ ***Substitution of data points***

```
R> ## generating substitute input data for update
R> n.sb <- 2
R> sIn.sb <- matrix(runif(n.sb * m1@ds), nrow = n.sb)
R> fIn.sb <- list(f1 = matrix(runif(n.sb * 10), ncol = 10),
+                 f2 = matrix(runif(n.sb * 22), ncol = 22))
R> ## generating substituting output data for updating
R> sOut.sb <- fgp_BB3(sIn.sb, fIn.sb, n.sb)
R> ## generating indices for substitution
R> ind.sb <- sample(1:(m1@n.tot), n.sb)
R> ## updating all, the scalar inputs, functional inputs and the output
R> m1up <- update(m1, sIn.sb = sIn.sb, fIn.sb = fIn.sb, sOut.sb = sOut.sb,
+                 ind.sb = ind.sb)

--------------
Update summary
--------------
* Complete tasks:
  - data substitution
```

Substituting points only from some of the data structures is also possible.

```
R> m1up1 <- update(m1, sIn.sb = sIn.sb, ind.sb = ind.sb) # scalar inputs

--------------
Update summary
--------------
* Complete tasks:
  - data substitution

R> m1up2 <- update(m1, sOut.sb = sOut.sb, ind.sb = ind.sb) # the output

--------------
Update summary
--------------
* Complete tasks:
  - data substitution
```

◇ ***Substitution of hyperparameters***

```
R> ## defining hyperparameters for substitution
R> var.sb <- 3
R> ls_s.sb <- c(2.44, 1.15)
R> ls_f.sb <- c(5.83, 4.12)
```

```
R> ## updating the model
R> m1up <- update(m1, var.sb = var.sb, ls_s.sb = ls_s.sb, ls_f.sb = ls_f.sb)

--------------
Update summary
--------------
* Complete tasks:
  - var substitution
  - scalar length-scale substitution
  - functional length-scale substitution
```

Substituting only one of the three data structures is possible as well.

```
R> m1up <- update(m1, var.sb = var.sb) # the variance

--------------
Update summary
--------------
* Complete tasks:
  - var substitution

R> m1up <- update(m1, ls_f.sb = ls_f.sb) # functional length-scale

--------------
Update summary
--------------
* Complete tasks:
  - functional length-scale substitution
```

◇ ***Re-estimation of hyperparameters***

```
R> m1up <- update(m1, var.re = TRUE) # the variance

** Computing optimal variance...

--------------
Update summary
--------------
* Complete tasks:
  - var re-estimation

R> m1up <- update(m1, ls_s.re = TRUE) # scalar length-scale parameters

** Presampling...

** Optimising hyperparameters...

final  value 2.841058
converged
```

```
** Hyperparameters done!

--------------
Update summary
--------------
* Complete tasks:
  - scalar length-scale re-estimation

R> m1up <- update(m1, ls_s.re = TRUE, ls_f.re = TRUE) # all length-scale

** Presampling...
** Optimising hyperparameters...

final  value 2.841058
converged

** Hyperparameters done!

--------------
Update summary
--------------
* Complete tasks:
  - scalar length-scale re-estimation
  - functional length-scale re-estimation

R> m1up <- update(m1, var.re = TRUE, ls_s.re = TRUE,
+                 ls_f.re = TRUE) # all hyperparameters

** Presampling...
** Optimising hyperparameters...

final  value 2.841058
converged

** Hyperparameters done!

--------------
Update summary
--------------
* Complete tasks:
  - var re-estimation
  - scalar length-scale re-estimation
  - functional length-scale re-estimation
```

It is possible to request multiple update tasks in a single call to `update()`. When doing so, tasks will be performed in the following order:

data deletion/substitution → data addition → hyperparameter substitution/re-estimation

4. Customization of a `funGp` model

Without delving too deep in the technical details, this section covers the choice of a `funGp` model through its so-called structural parameters. This set of categorical features (e.g., type of kernel, projection method) can be given different values in order to produce alternative models departing from the same input-output data. The interested reader is referred to Betancourt *et al.* (2020b) for a formal and more detailed explanation of the underlying theory.

4.1. Kernel family

The selection of a suitable kernel function is something that naturally comes to mind when working with GP models. At this point, **funGp** offers the possibility to choose among the Gaussian, Matérn 5/2 and Matérn 3/2 kernels. This selection can be specified in the `fgpm()` call through the argument `kertype`, which might take a value among `"gauss"`, `"matern5_2"` and `"matern3_2"`. See for instance the example below for the Gaussian kernel.

```
R> n.tr <- 25
R> sIn <- expand.grid(x1 = seq(0, 1, length = sqrt(n.tr)),
+                     x2 = seq(0, 1, length = sqrt(n.tr)))
R> fIn <- list(f1 = matrix(runif(n.tr * 10), ncol = 10),
+              f2 = matrix(runif(n.tr * 22), ncol = 22))
R> sOut <- fgp_BB3(sIn, fIn, n.tr)
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, kerType = "gauss")

** Presampling...

** Optimising hyperparameters...

final  value -3.688915
converged

** Hyperparameters done!
```

By default, `fgpm()` uses the Matérn 5/2 function, popular in the machine learning community.

4.2. Projection method

In earlier sections we mentioned DR, the process of reducing the dimension of the data structures in such a way and extent that the model becomes significantly more tractable while most of its prediction power is retained. A common DR approach used on functional inputs is to project each input onto a space of lower dimension. This method requires the constitution of the basis functions defining the projection space for each input, which may come from diverse families including B-splines (De Boor 1978), PCA (Jolliffe 2002) and PLS (Papaioannou, Ehre, and Straub 2019), among the most popular ones. The B-splines and PCA bases are currently implemented in **funGp** for the projection of functional inputs. This option is accessible in `fgpm()` through the `f_basType` argument, which can be set to `"B-splines"` or `"PCA"`. When multiple functional inputs are handled, a custom basis can be selected for each of them. This is done by passing an `array` with the selection for each input:

```
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut,
+             f_basType = c("B-splines", "PCA"))

** Presampling...

** Optimising hyperparameters...

final  value 2.164196
converged

** Hyperparameters done!
```

If multiple functional inputs are provided, but a single `f_basType` value is specified, that selection is used for all the inputs. By default, all functional inputs use a B-splines basis.

4.3. Projection dimension

This parameter is complementary to the projection method discussed above. Ideally, the projection dimension must be set substantially lower than the original one, but not so low that too much information is lost. The selection for each input can be specified in the `fgpm()` call through the `f_pdims` argument. Possible values are all the integer numbers from 0 to the original dimension of each input, with 0 denoting no projection. If multiple functional inputs are used, an `array` can be used to specify the selection for each. Below is an example where the first input is not projected while the second one is projected onto a space of dimension 7.

```
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0, 7))

** Presampling...

** Optimising hyperparameters...

final  value 13.032659
converged

** Hyperparameters done!
```

If multiple functional inputs are passed, but a single `f_pdims` value is specified, that selection is used for all the inputs. By default, all functional inputs are projected to dimension 3.

4.4. Distance for functions

Kernel-based regression models require the computation of the distance between input coordinates in order to estimate the behavior of the output. This is the case of GP models, which use such distances to compute the correlation between pairs of observations. A set of scaling factors called length-scale coefficients are normally used to quantify the rate of change of the

output in terms of each input. For scenarios with only scalar inputs, the typical is to use one length-scale parameter per input, which yields the distance

$$\|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|_{L^2,\boldsymbol{\theta_s}} := \sqrt{\sum_{k=1}^{d_s} \frac{\left\|x^{(k)} - \tilde{x}^{(k)}\right\|^2}{\left(\theta_s^{(k)}\right)^2}}, \tag{4}$$

with $\|\cdot\|$ the L^2 norm for scalars (i.e., the absolute value), and $\boldsymbol{\theta_s} = \left(\theta_s^{(1)}, \ldots, \theta_s^{(d_s)}\right)^\top$ the vector of length-scale parameters for the scalar inputs.

When dealing with functional inputs, the norm $\|\cdot\|$ must be replaced by a suitable norm for functions. Two options are currently implemented in **funGp**, both based on a projection of each functional input of the form:

$$\Pi^{(k)}(f^{(k)})(t) = \sum_{r=1}^{p^{(k)}} \alpha_r^{(k)} B_r^{(k)}(t), \tag{5}$$

with $f^{(k)}$ a curve of the k-th functional input, with values $f^{(k)}(t)$, $t \in \mathcal{T}_k$, $B_r^{(k)}$ the r-th basis function used for its projection, $\alpha_r^{(k)}$ the corresponding projection coefficient, and $p^{(k)}$ the projection dimension. Note that $p^{(k)}$ is typically significantly smaller than the number of time instants $N^{(k)}$ (see Section 3) and that when $p^{(k)} = N^{(k)}$, the function $f^{(k)}$ and its projection coincide at the time instants. Note that with the PCA projection method, the basis functions $B_1^{(k)}, \ldots, B_{p^{(k)}}^{(k)}$ are taken from the $N^{(k)}$ ordered principal component functions, and thus for instance $B_1^{(k)}$ does not depend on the choice of $p^{(k)}$. In contrast, with the B-spline basis functions, for each value of $p^{(k)}$, there is a specific set of $p^{(k)}$ basis functions (selected to cover the entire domain $\mathcal{T}_k$ approximately uniformly) and thus in particular $B_1^{(k)}$ changes when $p^{(k)}$ is changed, in general.

The first type of distance (that can also be thought of as a semi-norm) implemented for functions considers each curve as a whole element:

$$\|\boldsymbol{f} - \tilde{\boldsymbol{f}}\|_{G,\boldsymbol{\theta_f}} := \sqrt{\sum_{k=1}^{d_f} \frac{\int_{\mathcal{T}_k} \left(\sum_{r=1}^{p^{(k)}} \left(\alpha_r^{(k)} - \tilde{\alpha}_r^{(k)}\right) B_r^{(k)}(t)\right)^2 dt}{\left(\theta_f^{(k)}\right)^2}}, \tag{6}$$

with $\boldsymbol{f}$ and $\tilde{\boldsymbol{f}}$ two functional input points, $\mathcal{T}_k \subset \mathbb{R}$ the domain of $f^{(k)}$, and $\boldsymbol{\theta_f} = \left(\theta_f^{(1)}, \ldots, \theta_f^{(d_f)}\right)^\top$ the vector of length-scale parameters for the functional inputs. This distance is identified in the package as `L2_bygroup`, since it uses a single length-scale parameter for the group of projection terms linked to one functional input. Note that when $p^{(k)} = N^{(k)}$ as discussed above, the integral above is simply the square L^2 distance $\int [f^{(k)}(t) - \tilde{f}^{(k)}(t)]^2 dt$, thus making (6) similar to (4). Note also that the above integral is numerically handled as an average over the $N^{(k)}$ time instants.

The second type of distance implemented only considers the projection coefficients, and dis-

regards the basis functions:

$$\|\boldsymbol{f} - \tilde{\boldsymbol{f}}\|_{I,\dot{\boldsymbol{\theta}}_{\boldsymbol{f}}} := \sqrt{\sum_{k=1}^{d_f} \sum_{r=1}^{p^{(k)}} \frac{\left(\alpha_r^{(k)} - \tilde{\alpha}_r^{(k)}\right)^2}{\left(\dot{\theta}_{f,r}^{(k)}\right)^2}}, \tag{7}$$

where $\dot{\boldsymbol{\theta}}_{\boldsymbol{f}} = (\dot{\theta}_{f,r}^{(k)})_{1 \leq r \leq p^{(k)}, 1 \leq k \leq d_f}$ denotes the vector of functional length-scale coefficients. This distance is identified in the package as `L2_byindex` since it involves one length-scale parameter per projection index. Using this distance corresponds to a common approach, which is to first perform the projection and then use each projection coefficient as an individual scalar input of the model. Remark that if for each k the basis functions $B_1^{(k)}, \ldots, B_{p^{(k)}}^{(k)}$ are orthonormal and $\theta_f^{(k)} = \dot{\theta}_{f,1}^{(k)} = \cdots = \dot{\theta}_{f,p^{(k)}}^{(k)}$, then (6) and (7) coincide.

Below we show some examples on how to specify the distance for each functional input in the `fgpm()` call. For these examples we still consider two functional inputs, $f^{(1)}$ and $f^{(2)}$, with original dimensions 10 and 22, respectively:

```
R> n.tr <- 25
R> sIn <- expand.grid(x1 = seq(0, 1, length = sqrt(n.tr)),
+                     x2 = seq(0, 1, length = sqrt(n.tr)))
R> fIn <- list(f1 = matrix(runif(n.tr * 10), ncol = 10),
+              f2 = matrix(runif(n.tr * 22), ncol = 22))
R> sOut <- fgp_BB3(sIn, fIn, n.tr)
```

For the first example, $f^{(1)}$ uses the `L2_byindex` distance and no projection, which yields 10 length-scale coefficients. On the other hand, $f^{(2)}$ uses the `L2_bygroup` distance, which yields a single length-scale coefficient regardless of the projection dimension.

```
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0, 5),
+             f_disType = c("L2_byindex", "L2_bygroup"))

** Presampling...

** Optimising hyperparameters...

final  value 26.082583
converged

** Hyperparameters done!
```

Now, in a second example, we use the `L2_bygroup` distance for both inputs, which yields 2 length-scale coefficients, regardless of the projection dimensions.

```
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0, 5),
+             f_disType = "L2_bygroup")

** Presampling...
```

```
** Optimising hyperparameters...

final  value 13.759879
converged

** Hyperparameters done!
```

Finally, we use the `L2_byindex` distance for both inputs, we use no projection for $f^{(1)}$ and projection dimension 5 for $f^{(2)}$. This yields a total of $10 + 5 = 15$ length-scale coefficients.

```
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0, 5),
+             f_disType = "L2_byindex")

** Presampling...

** Optimising hyperparameters...

final  value 28.570786
converged

** Hyperparameters done!
```

If no projection is requested for some input, both distances use the original data instead of the projection coefficients, and (6) uses the identity as the matrix of discretized basis functions.

5. Automatic structural configuration

The possibility to set up the structural parameters of the model naturally comes with the non-trivial question of how to choose a good structural configuration, namely: (i) the state of each input (active or inactive), (ii) the DR method (e.g., PCA, B-splines), (iii) projection dimension, (iv) distance function used for each functional input, and (v) the type of kernel for the model (e.g., Gaussian, Matérn 5/2). In this section we introduce the `funGp` model factory, a boosting feature that enables the automatic selection of structural parameters seeking for optimal predictive performance. This tool is accessible through the `fgpm_factory()` function, which enables the smart exploration of the solution space composed of all the structural parameter configurations available under `fgpm()`.

5.1. Ant Colony based model selection

At this point, **funGp** performs heuristic optimization of structural parameters supported on ACO-Gp, the Ant Colony based algorithm presented in Betancourt, Bachoc, Klein, and Gamboa (2020a). In ACO-Gp, virtual ants go through a decision network, selecting at each node the value for one structural parameter of the model (Figure 7). Each end-to-end path in the decision network provides a structural configuration, that is a selection of levels of kernel function, projection basis and distance type, listed in Sections 4.1, 4.2 and 4.4, as well as the state of each scalar and functional input in the model (active or inactive).

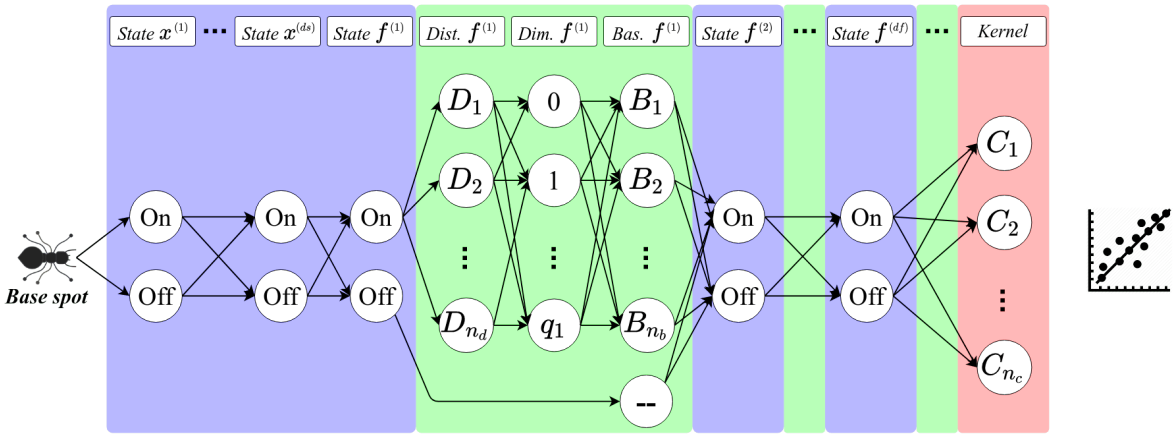


Figure 7: Decision network used by the ACO-Gp heuristic in **funGp**. *Dist.*, *Dim.*, and *Bas.*, stand for the distance, projection dimension and projection method (basis family) used for each functional input. The illustration considers d_s scalar inputs, d_f functional inputs, n_d types of distances, n_b projection methods and n_c kernel functions. The projection dimension 0 denotes no projection. Any end-to-end path over the network provides a candidate model.

The ACO-Gp algorithm is summarized as follows. Each edge in the decision network (a choice of the next node, see Figure 7) is initialized with a quantity (load) of pheromone. At each iteration, there is a population of individual ants, each of them going independently through the decision network. For each ant, each next edge is selected stochastically, an edge with higher pheromone load being selected with higher probability. Once all the ants have completed their network paths, each of these paths yields a quantitative model quality criterion (Q^2_{loocv} or Q^2_{hout}, see below). To complete the iteration, the pheromone load of each edge is updated. The pheromone loads of edges that belong to ant paths with better quality criteria are increased more. At the end of all the iterations, the ACO-Gp algorithm thus provides the list of all the structural configurations corresponding to each ant paths, associated with their quality criteria. The goal of this algorithm is that some of the configurations in the list have exceptionally good quality criteria. The calculations of the edge probabilities and pheromone updates can be tuned with parameters (see below), that enable to select a tradeoff between exploration (the edge probabilities are less dependent on the pheromone loads and the pheromone loads are more similar across edges) and exploitation (the converse).

Note that both ACO-Gp and its implementation within **funGp** are general enough to allow an easy extension to additional levels of kernel function, projection basis and distance type, to those listed in Sections 4.1, 4.2 and 4.4, in future versions of the package.

5.2. Getting started with the `funGp` model factory

The examples in this section are based on the analytic black-box function $\mathcal{H}$ from $[0,1]^5 \times \mathcal{F}$ to $\mathbb{R}$, with $\mathcal{F}$ the set of pairs of continuous functions $\boldsymbol{f} = (f^{(1)}, f^{(2)})^\top$ from $[0,1]$ to $\mathbb{R}$, defined

by

$$\mathcal{H}(\boldsymbol{x}, \boldsymbol{f}) = a + \left[b_0 + b_1 x^{(1)} + b_2 x^{(1)2} + b_3 x^{(2)} + b_4 x^{(3)}\right]^2 + c x^{(2)2} x^{(5)3} \cos x^{(1)} + d \sin x^{(4)} + e \sin x^{(4)} \times \int_0^1 (1-t)\, f^{(1)}(t) dt + \left[h_0 + h_1 x^{(1)} x^{(5)}\right] \times \int_0^1 t f^{(2)}(t) dt,$$

with $a = 10$, $b_0 = -6$, $b_1 = 5/\pi$, $b_2 = -5/4\pi^2$, $b_3 = 1$, $b_4 = 4$, $c = 10(1 - 1/8\pi)$, $d = -280\pi$, $e = 840\pi$, $h_0 = 240\pi$ and $h_1 = 3\pi$.

This function is adapted from the second analytic example of Muehlenstaedt *et al.* (2017). It is accessible in **funGp** through the black-box function `fgp_BB7()`. Here we generate the scalar and functional input values in a similar way to how we did in the previous sections:

```
R> set.seed(100)
R> n.tr <- 32
R> sIn <- expand.grid(x1 = seq(0,1,length = n.tr^(1/5)),
+                     x2 = seq(0,1,length = n.tr^(1/5)),
+                     x3 = seq(0,1,length = n.tr^(1/5)),
+                     x4 = seq(0,1,length = n.tr^(1/5)),
+                     x5 = seq(0,1,length = n.tr^(1/5)))
R> fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10),
+              f2 = matrix(runif(n.tr*22), ncol = 22))
R> sOut <- fgp_BB7(sIn, fIn, n.tr)
```

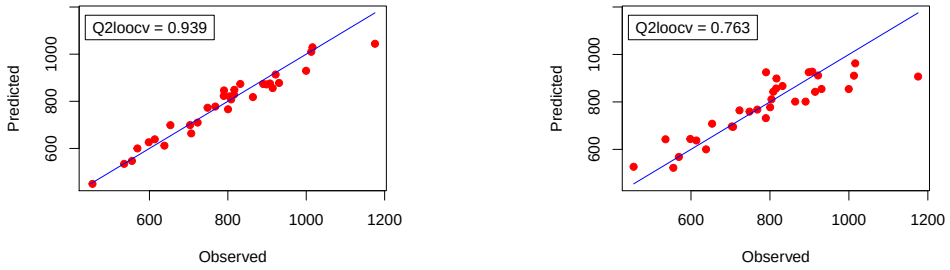
◇ ***Getting started***

In our first example, we make a basic call to the model factory using its default arguments:

```
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut) # (~10 seconds)

** Initializing decision network...

** Optimizing structural parameters...


** Ants are done ;)
```

The output of `fgpm_factory()` is an object of class `"Xfgpm"` containing the optimized functional model (i.e., an object of class `"fgpm"` as the one delivered by `fgpm()`), a statistical indicator of its prediction power, the selected structural configuration, a record of all the evaluated models, among various other types of information. For now, we focus on the `@model` slot storing the optimized regression model. Let us first compare this optimized model with the one produced by `fgpm()` using its default arguments. This can be done by means of the `plot()` function introduced in Section 3.1, which receives an `fgpm` object as argument.

```
R> ## optimized model
R> plot(xm@model, main = "")

R> ## model with default fgpm() structural configuration
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut)
```



(a) Model produced by `fgpm_factory()` using its default arguments.

(b) Model produced by `fgpm()` using its default arguments.

Figure 8: Calibration plot for optimized and unoptimized structural configurations.

```
** Presampling...

** Optimising hyperparameters...

final  value 205.640552
converged

** Hyperparameters done!
```

```
R> plot(m1, main = "")
```

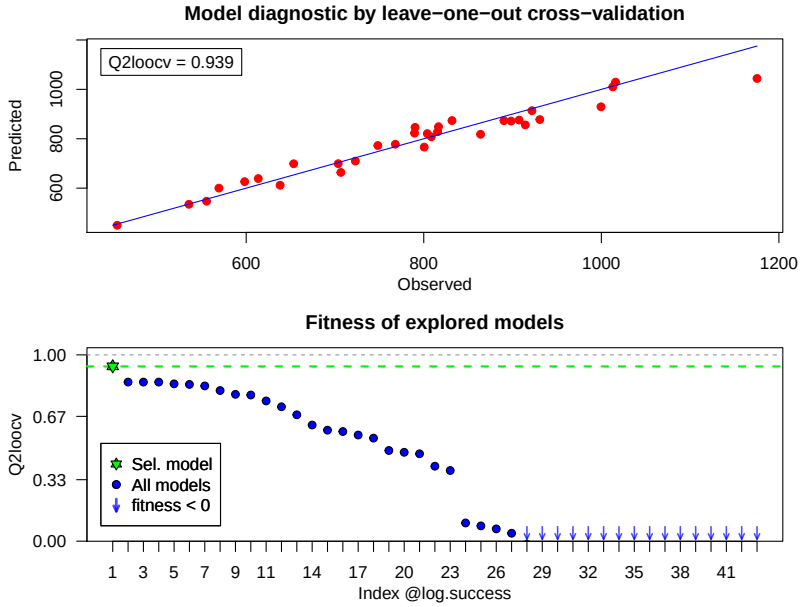
The outputs of the two `plot()` calls are displayed in Figures 8a and 8b, respectively. As can be seen, right away, just by calling `fgpm_factory()` with its default arguments, we were able to find a model of greater quality than the one obtained with the default `fgpm()` arguments. Some key points in the light of this first result are:

- While the default structural parameters in `fgpm()` provide reasonable predictions, they are not tailored to the data set at hand. In that view, the purpose of having `fgpm_factory()` in the package is to be able to generate high quality structural configurations tailored to any data set that `fgpm()` could handle.
- The superiority of the model delivered by `fgpm_factory()` is exclusively fostered by the optimization of the structural parameter configuration, and is unrelated to the mechanism for the optimization of the hyperparameters. Each model evaluated by `fgpm_factory()` is internally created by a call to `fgpm()`. Thus, *the same mechanism of hyperparameter optimization is used by both functions.*

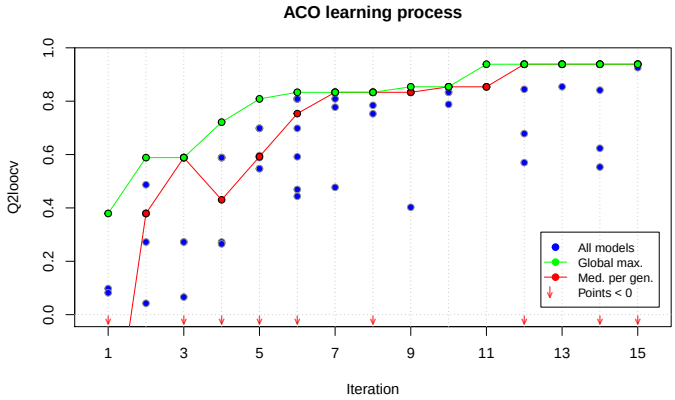
Let us move on with the exploration of the `Xfgpm` object delivered by `fgpm_factory()` using the `plot()` method.

```
R> plot(xm, which = "diag")
```

```
R> plot(xm, which = "evol")
```



(a) Absolute and relative quality diagnostic plots delivered by `plot()` with `which = "diag"` (default value).



(b) Model quality evolution plot delivered by `plot()` with `which = "evol"`.

Figure 9: Figures available for the `Xfgpm` model delivered by `fgpm_factory()`.

The `plot()` method has an argument `which` allowing to choose between a diagnostic plot and an evolution plot. With the default choice `which = "diag"` the plot provides information on the absolute and relative quality of the selected model (Figure 9a). The choice `which = "evol"` illustrates the evolution in quality of the explored models along the optimization (Figure 9b). In the evolution plot, the points below zero usually correspond to models whose hyperparameters were difficult to optimize. This happens sporadically during the log-likelihood optimization for GPs due to the non-linearity of the objective function, and is not an issue that affects exclusively the **funGp** package.

The quality of the selected model generally increases with the number of iterations completed by the algorithm. By default, the model factory performs 15 iterations. This quantity can be

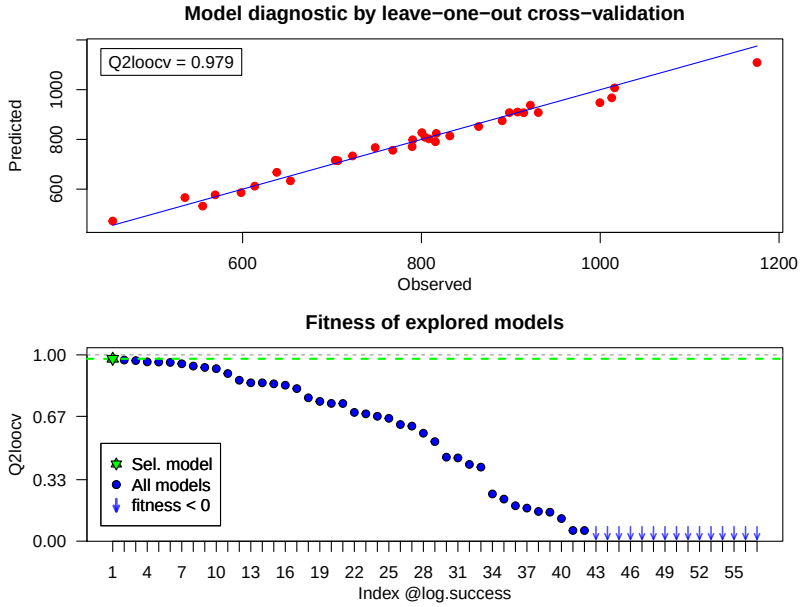


Figure 10: Improved quality of the optimized model using a larger number of iterations.

customized through the `setup` argument of the function, defined as a `list`. The way to do it is by including an element named `n.iter` in the `setup list` and giving it the value of the required number of iterations.

```
R> set.seed(100)
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut,
+    setup = list(n.iter = 30)) # calling the funGp factory (~6.5 seconds)

** Initializing decision network...

** Optimizing structural parameters...


** Ants are done ;)
```

Figure 10 displays the output of the `plot()` call on the model obtained with 30 iterations. In this case, the algorithm explored a larger number of model configurations and was able to find a model of higher quality than the one obtained with the default 15 iterations (Figure 9).

In the examples above, `fgpm_factory()` optimized the model structure for Q^2_{loocv} (see Section 3). Validating against external observations is also possible. This type of optimization can be requested by specifying, through the `ind.vl` argument, the indices of the data that should be used for validation. For instance, suppose that we have the same data as in the previous example, but we want to use about 85% of the points for training and the remaining ones for validation.

```
R> ## about 15% of points for validation
R> ind.vl <- sample(seq_len(n.tr), 5)
```

```
R> ## calling the funGp factory (~2 seconds)
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, ind.vl = ind.vl)

** Initializing decision network...

** Optimizing structural parameters...


** Ants are done ;)
```

With this call, the model factory trains each model using all the data except for the points specified by `ind.vl`. Once built, each model is used to predict the output at the points ignored during training, and the predictive squared correlation coefficient Q^2_{hout} (Nilsson *et al.* 1997) is computed. This procedure ensures homogeneity in the comparison, since all the models use the same training and validation sets. To account for the sampling noise, it is possible to define multiple pairs of training-validation sets. This option is requested to the factory by passing a `matrix` instead of an `array` through the argument `ind.vl`. Such a `matrix` should have a vector of validation indices per column, and as many columns as replicates.

```
R> ## ~15% of points for validation, 30 replicates
R> ind.vl <- replicate(30, sample(seq_len(n.tr), 5))
R> ## calling the funGp factory (~4 minutes)
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, ind.vl = ind.vl)
```

Note that the calibration plot produced by `plot()` with `which = "diag"` will always report the Q^2_{loocv} statistic, regardless of whether this or the Q^2_{hout} is used for the optimization of the structural parameters. In contrast, the evolution plot produced by this same function will display the statistic used during the optimization. When validation indices are provided, the optimized model stored in the `Xfgpm` object is one trained with as many points as left once the specified validation points are removed. When multiple validation sets are specified, the optimal model is selected by first identifying the structural configuration with largest average Q^2_{hout} over all the replicates, and then picking among the corresponding models the one with largest individual Q^2_{hout}.

5.3. Setting up the parameters of the ACO algorithm

Our model selection algorithm relies on a set of parameters typical of any Ant Colony based method (recall the description of the method in Section 5.1). Those parameters control the number of individuals and iterations, degree of exploration, rate of convergence, as well as the learning mechanism in the algorithm. The full list of parameters implemented in the `fgpm_factory()` function are listed below; those marked with an asterisk (*) are explained more thoroughly in Betancourt *et al.* (2020a).

Initial pheromone load: at the beginning of the algorithm, each edge in the decision network is assigned a quantity indicating its desirability for the ants.

- `tao0`: initial pheromone load on links pointing out to the selection of a distance type, a projection basis or a kernel type. `Default: 0.1`.

- `dop.s`: factor to control how likely it is to activate a scalar input. It operates on a relation of the type `A = dop.s * I`, where `A` is the initial pheromone load of links pointing out to the activation of scalar inputs and `I` is the initial pheromone load of links pointing out to their inactivation (which is set equal to `tao0`). `Default: 1`.
- `dop.f`: analogous to `dop.s` for functional inputs. `Default: 1`.
- `delta.f` and `dispr.f`: shape parameters for the regularization function that determines the initial pheromone values on the links connecting the `L2_byindex` distance with the projection dimension*. `Default: 2` and `1.4`, respectively.

Local pheromone update: as the ants traverse the decision network, the pheromone load of the used links is reduced to foment diversification in the structural configurations generated.

- `rho.l`: pheromone evaporation rate*. `Default: 0.1`.

Global pheromone update: the algorithm works by iterations, each involving a given number of ants. After each iteration, an update process is performed to increase the pheromone load in the links included in the best configurations evaluated so far.

- `n.ibest`: the algorithm always reinforces the links of the best `n.ibest` ants of the current iteration; how many ants should be used? `Default: 1`.
- `u.gbest`: should the links of the best ant over all the iterations so far be reinforced during the global pheromone update as well? `Default: FALSE`.
- `rho.g`: learning reinforcement rate*. `Default: 0.1`.

Population factors: the extent of the search is primarily driven by the number of iterations and ants per iteration.

- `n.iter`: number of iterations. Each iteration involves exploration, local pheromone update, constitution of model configurations, evaluation of performance in prediction, and system feedback through global pheromone update (recall Section 5.1). `Default: 15`.
- `n.pop`: number of ants (structural configurations) per iteration. `Default: 10`.

Bias strength: ants use one of two rules to select their next node at each step. The first rule leads the ant through the link with larger pheromone load; the second rule works based on probabilities which are proportional to the pheromone load on the feasible links. The ants will randomly chose one of the two rules at each time.

- `q0`: at each step, each ant will opt for rule 1 with probability `q0`*. `Default: 0.95`. For larger number of input variables, we recommend to slightly reduce it to e.g., `0.90`. This might promote the evaluation of each input in at least some few models.

The default values of ACO-Gp parameters in **funGp** are based on the setup used by Dorigo and Gambardella (1997) in the introductory paper of the Ant Colony System. Similar values have proved to deliver a good performance for ACO-based algorithms in a variety of applications (see e.g., Li, Soleimani, and Zohal (2019) or Singh, Singh, Kumar, and Biswas 2020). All the parameters listed above can be customized in a `fgpm_factory` call through the argument `setup`, which should be a `list`. Below is an example using arbitrary setup values.

```
R> ## custom heuristic configuration
R> mysup <- list(tao0 = .15, dop.s = 1.2, dop.f = 1.3, delta.f = 4,
+                dispr.f = 1.1, rho.l = .2, u.gbest = TRUE, n.ibest = 2,
+                rho.g = .08, n.iter = 30, n.pop = 12, q0 = .85)
R> ## calling the funGp factory (~18 seconds)
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, setup = mysup)

** Initializing decision network...

** Optimizing structural parameters...


** Ants are done ;)
```

5.4. Defining the solution space

By default, `fgpm_factory()` considers feasible all possible combinations of: inputs state, distance type, projection dimension, basis family, and kernel family. However, the user is allowed to modify the solution space by imposing a system of constraints. This is achieved through the `ctraints` argument, which should be provided as a `list`. Below is an example considering the following list of constraints:

- Keep both scalar inputs always active;
- Keep $f^{(2)}$ always active;
- Only use the `L2_byindex` distance for $f^{(2)}$;
- Input $f^{(2)}$ should be projected onto a space of dimension 4;
- Input $f^{(1)}$ should be projected onto a space of dimension ≤ 5;
- Only use the B-splines projection method for $f^{(1)}$;
- Only test the Matérn 5/2 and Gaussian kernels.

```
R> ## custom constraints
R> myctr <- list(s_keepOn = c(1, 2), f_keepOn = c(2),
+                f_disTypes = list("2" = c("L2_byindex")),
+                f_fixDims = matrix(c(2, 4), ncol = 1),
+                f_maxDims = matrix(c(1,5), ncol = 1),
+                f_basTypes = list("1" = c("B-splines")),
+                kerTypes = c("matern5_2", "gauss"))
R> ## calling the funGp factory (~15 seconds)
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, ctraints = myctr)
```

The consistency of the explored configurations with the specified constraints can be verified through the `@log.success@sols` slot of the `Xfgpm` object returned by `fgpm_factory()`. This slot provides a `data.frame` with all the configurations successfully built and tested. Their corresponding prediction quality indicators can be obtained from the `@log.success@fitness` slot of the `Xfgpm` object (either Q^2_{loocv} or Q^2_{hout}, depending on the optimized statistic). This is summarized by the `summary()` function as follows.

```
R> summary(xm)
```

5.5. Time based stopping condition

In practice, it might be difficult to know in advance how many iterations will be enough to find a good configuration. Hence, `fgpm_factory()` offers an alternative stopping condition based on processing time. This feature is accessible in `fgpm_factory()` through the `time.lim` argument, which should be provided in seconds.

```
R> mysup <- list(n.iter = 2000) # a sufficiently large number of iterations
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, setup = mysup,
+                     time.lim = 60)

** Initializing decision network...

** Optimizing structural parameters...

** Time limit reached, exploration stopped after 60.01 seconds.


** Ants are done ;)
```

The number of iterations was set large enough to avoid a premature stop of the algorithm. Once the time limit is reached, the algorithm will attempt to stop as soon as possible, however, an ongoing model training process (i.e., the hyperparameter optimization) will never be interrupted. Thus, the actual processing time will often slightly exceed the specified budget.

5.6. Further exploring the `Xfgpm` object

We close this section by discussing in more detail the information contained in the `Xfgpm` object delivered by `fgpm_factory()`. Below we provide a list of the slots included in this object.

Optimal structural configuration

- `@model`: object of class `"fgpm"` associated to the selected model configuration.
- `@structure`: `data.frame` indicating the selected structural configuration.
- `@stat` and `@fitness`: type (Q^2_{loocv} or Q^2_{hout}) and value of the performance statistic used for the optimization of the structural parameters.

Record of explored models

- `@log.success`: object of class `"antsLog"` with the structure, function call and performance statistic of all the models successfully built and tested during the optimization.
- `@log.crashes`: object of class `"antsLog"` with the structure and function calls of all the models whose `fgpm()` function call crashed.

Exploration extent

- `@n.solspace`: number of feasible structural configurations.

- `@n.explored`: number of structural configurations successfully built and tested during the optimization.

Further information

- `@details`: a `list` containing: (i) the set of heuristic parameters used; and (ii) the series of fitness vectors over the iterations of the heuristic.
- `@factoryCall`: a reminder of the expression used in the `fgpm_factory()` call.

The `@log.success` slot opens the possibility for a variety of interesting analyses that could be performed after the structural configuration. Based on the `@log.success@sols` component, for instance, which contains a `data.frame` with the structural parameters used in each explored model, one could seek to identify patterns leading to high quality models. The `@log.success@args` component contains analogous information, but in a format that allows the easy reconstruction of any explored model. We illustrate this possibility with the following example. We start from the structural optimization using 32 training points.

```
R> set.seed(100)
R> n.tr <- 32
R> sIn <- expand.grid(x1 = seq(0, 1, length = n.tr^(1/5)),
+                     x2 = seq(0, 1, length = n.tr^(1/5)),
+                     x3 = seq(0, 1, length = n.tr^(1/5)),
+                     x4 = seq(0, 1, length = n.tr^(1/5)),
+                     x5 = seq(0, 1, length = n.tr^(1/5)))
R> fIn <- list(f1 = matrix(runif(n.tr * 10), ncol = 10),
+              f2 = matrix(runif(n.tr * 22), ncol = 22))
R> sOut <- fgp_BB7(sIn, fIn, n.tr)
R> xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut) # (~10 seconds)

** Initializing decision network...

** Optimizing structural parameters...


** Ants are done ;)
```

Let us now rebuild the three best models using a larger number of training points. The `modelDef()` function returns a parsed R code that can be evaluated to rebuild the model corresponding to an index given by the `ind` formal argument. This code is to be evaluated in an environment containing the required objects `sIn`, `fIn` and `sOut`. By default, `eval()` uses the global environment.

```
R> n.tr <- 243
R> sIn <- expand.grid(x1 = seq(0, 1, length = n.tr^(1/5)),
+                     x2 = seq(0, 1, length = n.tr^(1/5)),
+                     x3 = seq(0, 1, length = n.tr^(1/5)),
+                     x4 = seq(0, 1, length = n.tr^(1/5)),
+                     x5 = seq(0, 1, length = n.tr^(1/5)))
```

```
R> fIn <- list(f1 = matrix(runif(n.tr * 10), ncol = 10),
+              f2 = matrix(runif(n.tr * 22), ncol = 22))
R> sOut <- fgp_BB7(sIn, fIn, n.tr)
R> m1 <- eval(modelDef(xm, ind = 1))

** Presampling...

** Optimising hyperparameters...

** Hyperparameters done!

R> m2 <- eval(modelDef(xm, ind = 2))

** Presampling...

** Optimising hyperparameters...

** Hyperparameters done!

R> m3 <- eval(modelDef(xm, ind = 3))

** Presampling...

** Optimising hyperparameters...

** Hyperparameters done!
```

Remark that `m1`, `m2` and `m3` are rebuilt from a different data set from the original data set used to obtain them when building `xm`. In the case where these three models were rebuilt using the same data set as when building `xm`, the estimated hyperparameters could still take different values than when building `xm`, since the hyperparameter optimization precedures use random initializations.

The three models `m1`, `m2` and `m3` can easily be used to make predictions thanks to the `get_active_in()` function, which automatically removes the inactive inputs from prediction or simulation data structures based on the `fgpm()` arguments of the model. To illustrate this, we start by stacking the three rebuilt models and their arguments into two `lists`.

```
R> modStack <- list(m1, m2, m3)
R> argStack <- xm@log.success@args[1:3]
```

Then, we generate some input points for prediction.

```
R> n.pr <- 32
R> sIn.pr <- expand.grid(x1 = seq(0, 1, length = n.pr^(1/5)),
+                        x2 = seq(0, 1, length = n.pr^(1/5)),
+                        x3 = seq(0, 1, length = n.pr^(1/5)),
+                        x4 = seq(0, 1, length = n.pr^(1/5)),
+                        x5 = seq(0, 1, length = n.pr^(1/5)))
R> fIn.pr <- list(f1 = matrix(runif(n.pr * 10), ncol = 10),
+                 f2 = matrix(runif(n.pr * 22), ncol = 22))
```

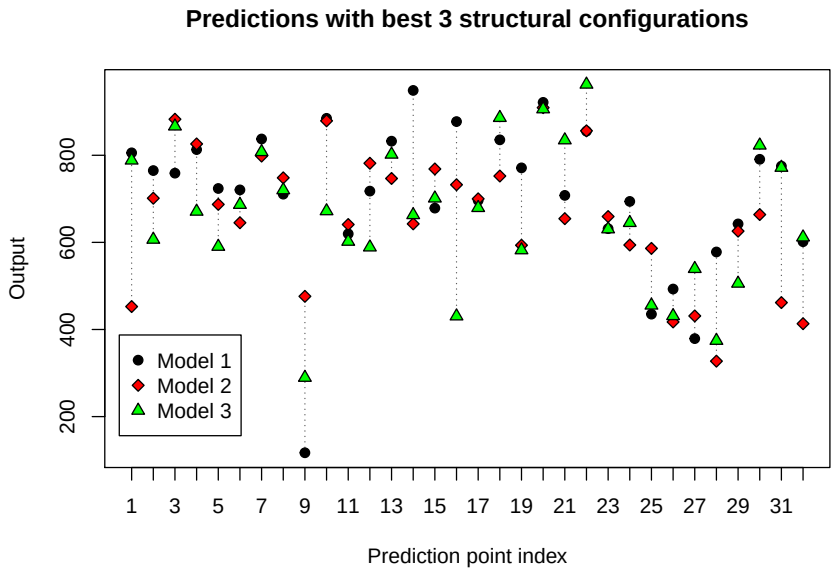


Figure 11: Predictions with models based on the best three structural configurations.

Finally, we perform predictions based on the three models with the aid of `get_active_in()`.

```
R> preds <- do.call(cbind, Map(function(model, args) {
+              active <- get_active_in(sIn = sIn.pr, fIn = fIn.pr, args)
+              predict(model, sIn.pr = active$sIn.on,
+                              fIn.pr = active$fIn.on)$mean
+        }, modStack, argStack))
```

The resulting predictions are displayed in Figure 11.

6. Parallelization in funGp

Both, the `fgpm()` and `fgpm_factory()` functions have been equipped with the ability to exploit the availability of parallel environments. Now we explain how to access this feature.

6.1. Parallelized hyperparameter optimization

In **funGp**, the calibration of the hyperparameters of the model is made by likelihood maximization. For GPs, this corresponds to a nonlinear optimization problem, sometimes strongly affected by the selection of the starting points. A common way to deal with this issue is to start the optimization multiple times from different points, which prevents the stagnation at local optima. This option is accessible in `fgpm()` through the argument `n.starts`, which should be assigned an integer value corresponding to the number of starting points to use. Below is an example using 10 starting points.

```
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, n.starts = 10)

** Presampling...
```

```
** Optimising hyperparameters...

** Parallel backend register not found. Multistart optimizations done in sequence.

** Hyperparameters done!
```

Since each starting point triggers an independent optimization process, the requested task can be performed in parallel. To do so, the user must define a parallel processing cluster and then pass it to `fgpm()` through the `par.clust` argument. As a good practice, the cluster must be stopped right after finishing the requested task in order to prevent memory issues.

```
R> cl <- parallel::makeCluster(3)
R> m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut,
+             n.starts = 10, par.clust = cl)

** Presampling...

** Optimising hyperparameters...

** Parallel backend register found. Multistart optimizations done in parallel.

final  value 1107.918763
converged
final  value 1107.918763
converged
final  value 1107.918763
converged
final  value 1107.918763
converged
final  value 1107.918763
converged
iter   10 value 1107.918763
final  value 1107.918763
converged
final  value 1107.918763
converged
final  value 1107.918763
converged
final  value 1107.918763
converged
final  value 1107.918763
converged

** Hyperparameters done!

R> parallel::stopCluster(cl)
```

6.2. Parallelized structural optimization

During a call to `fgpm_factory()`, each time all the ants of one iteration complete a feasible path, the corresponding models are built and evaluated in an independent fashion. Thus, this task can be performed in parallel. The way to request this to `fgpm_factory()` is identical to how it is requested to `fgpm()`. The user must define a parallel processing cluster and then pass it to `fgpm_factory()` through the `par.clust` argument, as shown below.

```
R> cl <- parallel::makeCluster(3)
R> xm.par <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, par.clust = cl)
R> parallel::stopCluster(cl)
```

7. Flooding real case application

Here we apply the functionalities of **funGp** to a real case that is related to the problem of setting up a rapid FEWS for coastal flooding at Gâvres, France (Idier *et al.* 2021). The modeling of coastal flooding at the proper resolution (metric) to predict floods in urban areas and account precisely for processes such as wave overtopping is so time consuming (i.e., the computation time is hardly smaller than the real time) that forecasting efforts are greatly hindered. To overcome this difficulty, one solution relies on the construction of metamodels by following the approach described by (Betancourt *et al.* 2020b).

We focus here on the prediction of the maximum value of the flooded area over time (denoted Y). See histogram in Figure 12.

```
R> set.seed(100)
R> load("Replication_data/FloodingCase.RData")
R> df <- length(Xf) ## Number of functional inputs
R> n <- nrow(Xs)
R> hist(Y, xlab = "Flooded Area (m^2)", main = "Model outputs")
```

Seven offshore meteo-oceanic forcing conditions depicted in Figure 13 are considered, namely

- the tide;
- the surge;
- the significant wave height, denoted H_{sx} and H_{sy} (projected onto the Cartesian x- and y-axis by means of the wave direction);
- the peak period of waves;
- the wind speed at 10m height, denoted U_x and U_y (projected onto the Cartesian x- and y-axis by means of the wind direction).

Each time series is sampled every 10 minutes (37 time steps) over the time interval (HT - 3 hours to HT + 3 hours), with HT referring to high tide.

To ease the analysis, the time-varying offshore meteo-oceanic forcings have been decomposed as $A_0 + f_n(t)$ where A_0 is a constant value over time (corresponding to the tide peak, i.e., the

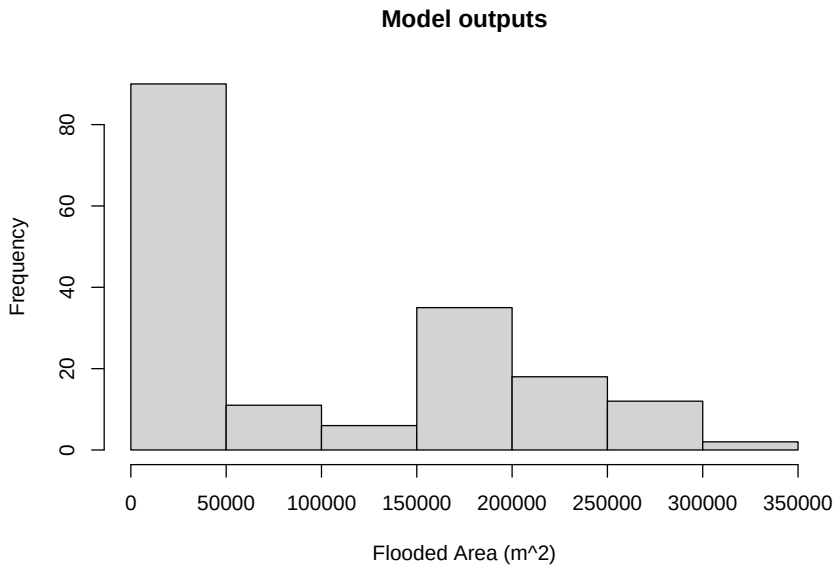


Figure 12: Histogram of flooded area (m^2).

maximum value of the tide over the considered time interval, and to the temporal average for the other variables), and $f_n(t)$ is a time-varying signal after removal of A_0.

In total the inputs of the GP model include seven $f_n(t)$ time series, eight scalar inputs (the seven A_0 scalar inputs as well as one additional scalar input corresponding to the mean sea level msl).

The random selection of the inputs is based on the procedure described by Idier *et al.* (2021, Sect. 2.5.2). A total number of 174 numerical experiments were performed. To showcase **funGp** functionalities, we use 100 numerical experiments for the training and 74 for the predictions.

In this example, we set up a GP model with default parameters i.e., a `"matern5_2"` kernel, a `"L2_bygroup"` distance to be used for each functional coordinates within the covariance function of the GP, and a projection using B-splines basis functions onto a 3-dimensional space.

```
R> ## definition of the training and test samples
R> id.train <- sample(1:n, 100, replace = FALSE)
R> id.test <- (1:n)[-id.train]
R>
R> Xs.train <- Xs[id.train, ]
R> Xs.test <- Xs[id.test, ]
R>
R> Xf.train <- Xf.test <- list()
R> for (i in 1:df){
+    Xf.train[[i]] <- Xf[[i]][id.train, ]
+    Xf.test[[i]] <- Xf[[i]][id.test, ]
+  }
R> Y.train <- Y[id.train]
R> Y.test <- Y[id.test]
```

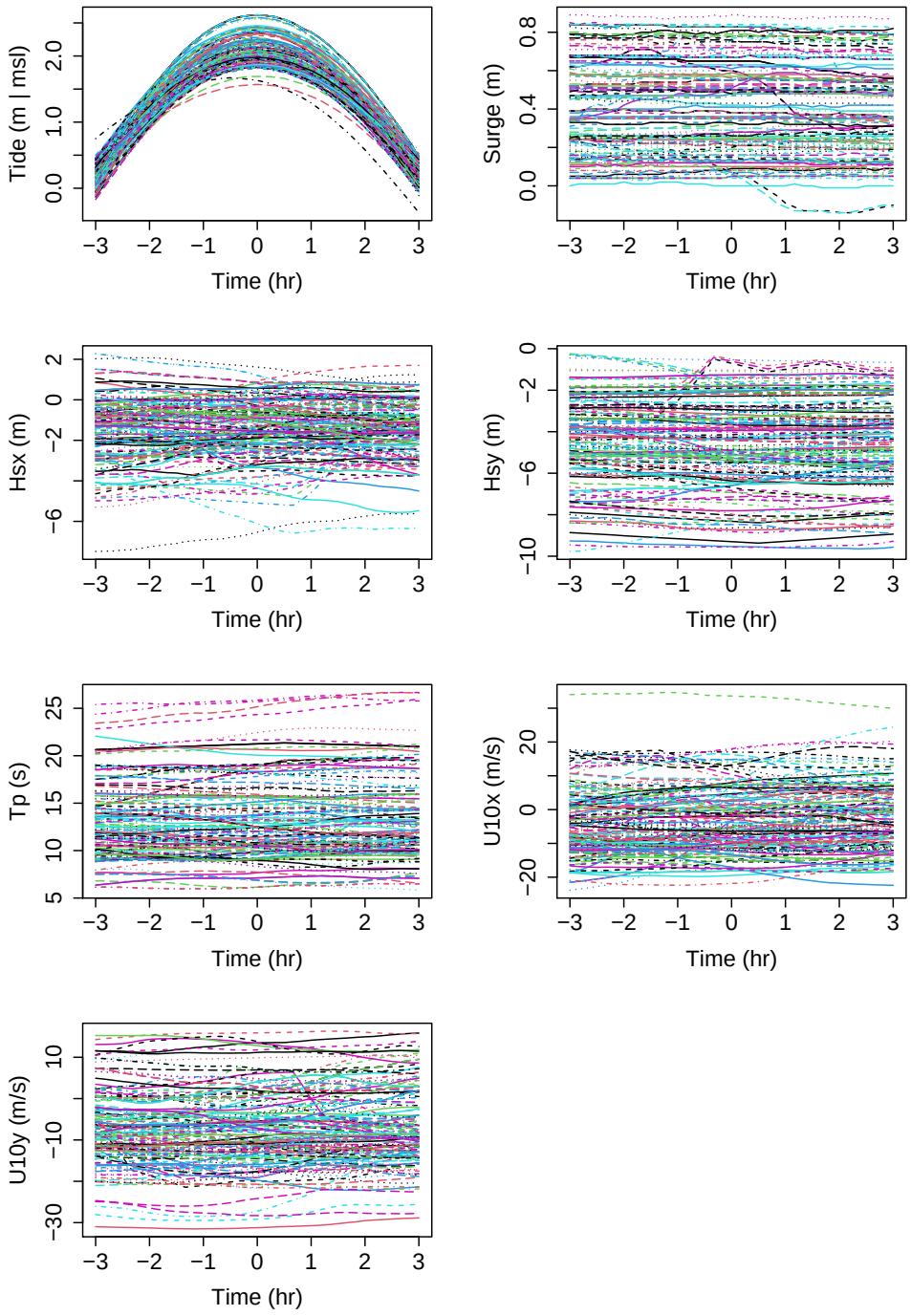


Figure 13: Functional inputs of the flooding case.

```
R>
R> ## fitting
R> m1 <- fgpm(sIn = Xs.train, fIn = Xf.train, sOut = Y.train)


** Presampling...
```

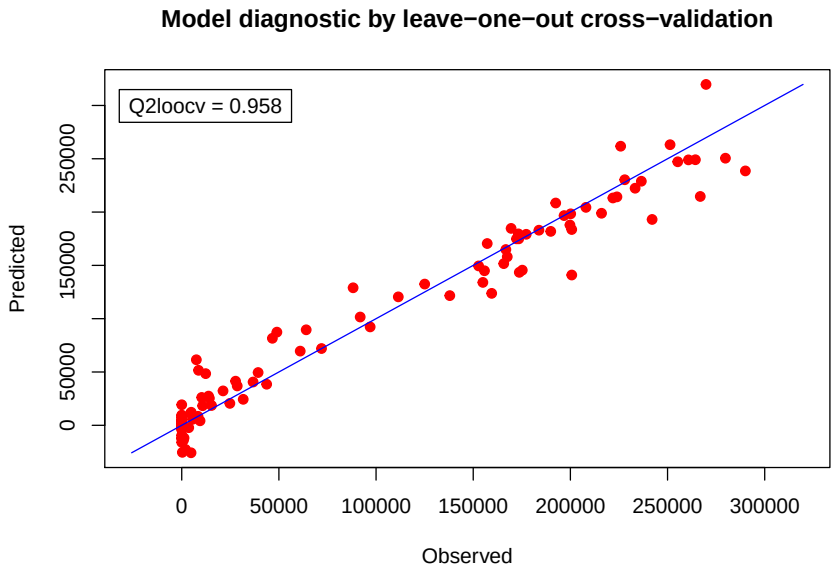


Figure 14: Leave-one-out cross validation diagnostic for the flooding application case.

```
** Optimising hyperparameters...

iter   10 value 1182.014146
iter   20 value 1178.110135
iter   30 value 1173.998820
iter   40 value 1172.713992
iter   50 value 1171.349083
iter   60 value 1169.167690
iter   70 value 1169.000458
iter   80 value 1168.063292
iter   90 value 1167.251745
iter  100 value 1167.098221
final  value 1167.042455
stopped after 101 iterations

** Hyperparameters done!
```

We first check the predictive capability of the constructed GP model using leave-one-out cross-validation. Figure 14 compares the observed and predicted Y values and shows a very satisfactory agreement with a Q^2_{loocv} >0.95.

```
R> plot(m1)
```

An alternative option is to check the predictive capability by means of a dataset of independent model results (Figure 15). The predictions can be performed by using the kriging mean plotted together with the prediction intervals at 0.95.

```
R> m1.preds <- predict(m1, sIn.pr = Xs.test, fIn.pr = Xf.test)
R> plot(m1.preds)
```

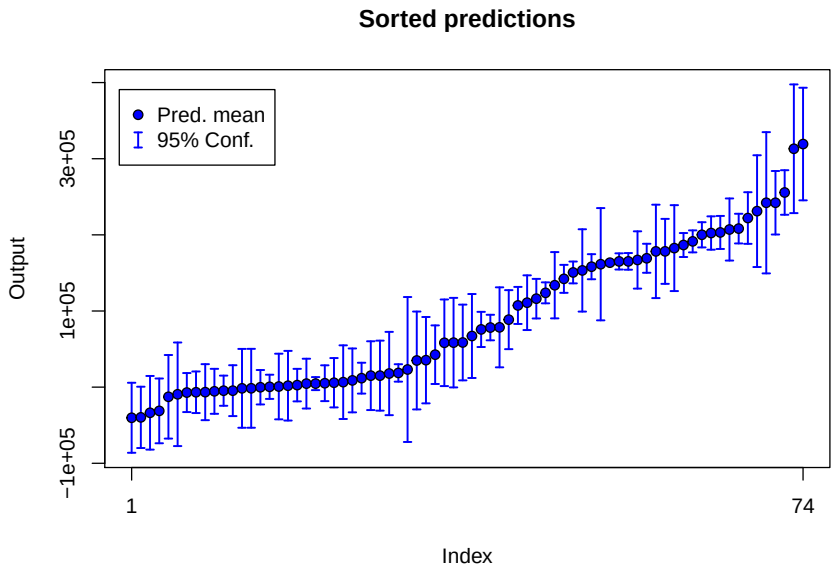


Figure 15: Predictions on independent samples for the flooding application case.

Due to the many assumptions (related to the account for 8 scalar and 7 functional inputs) to be made to parameterize the GP model, we apply the ACO-Gp algorithm (with default parameters) to automatically select the structural parameters depending on the predictive performance. Interestingly, the examination of the best five solutions (with Q^2_{loocv} ranging from 0.952 to 0.964, see Figure 16) indicates that the wind characteristics (scalar inputs denoted X7 and X8, and functional inputs denoted F6 and F7) are not included in the structure of the GP models. These models majoritarily include four time series, namely the tide, H_{sx}, H_{sy} and T_p. The next line of code was executed to call the model factory, and it is here commented because its execution time was significant.

```
R> ## xm <- fgpm_factory(sIn = Xs.train, fIn = Xf.train, sOut = Y.train)
R> # plotting the solution
R> plot(xm, which = "diag")

R> # 6 best solutions
R> ## head(xm@log.success@sols, "Q2" = xm@log.success@fitness)
R> summary(xm, n = 5)

State of inputs
  X1 X2 X3 X4 X5 X6 X7 X8 F1 F2 F3 F4 F5 F6 F7  Kern    Q2
1  x  x  x  x  x  x        x     x  x  x       mat32 0.964
2  x  x  x  x  x  x        x     x  x  x       mat32 0.963
3  x  x  x  x  x  x        x     x     x       mat32 0.961
4  x  x  x  x  x  x        x     x  x  x       mat32 0.957
5  x  x  x  x     x        x     x  x          mat32 0.952
Details for functional inputs
  F1 D_F1 Dim_F1 Bas_F1 F2 D_F2 Dim_F2 Bas_F2 F3 D_F3 Dim_F3 Bas_F3
1  x  grp     29    PCA      --      -     --  x  grp     23   Bspl
```

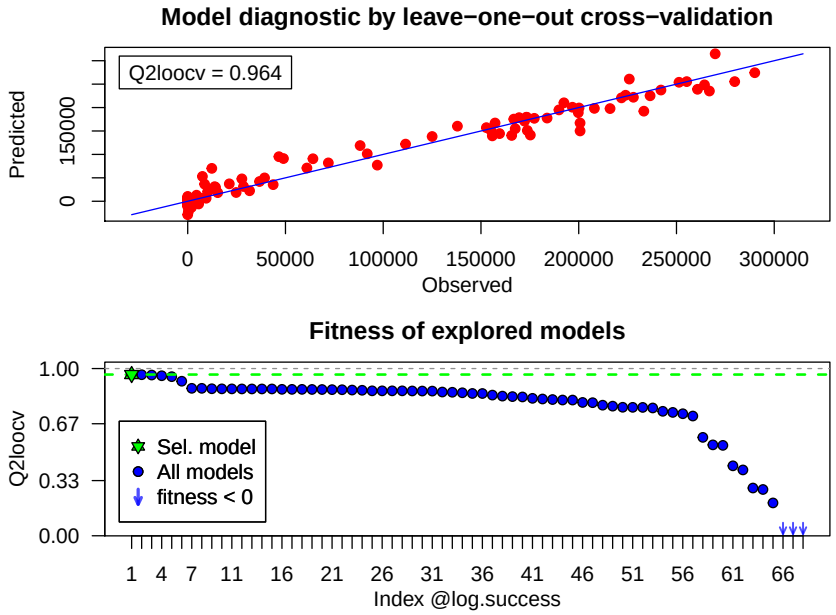


Figure 16: Best solutions resulting from the ACO-Gp algorithm applied to the flooding case.

```
2  x  idx      1   Bspl      --      -     --  x  grp     30   Bspl
3  x  idx      1   Bspl      --      -     --  x  grp     23    PCA
4  x  grp     29    PCA      --      -     --  x  idx      2   Bspl
5  x  grp     29    PCA      --      -     --  x  grp     23   Bspl
  F4 D_F4 Dim_F4 Bas_F4 F5 D_F5 Dim_F5 Bas_F5 F6 D_F6 Dim_F6 Bas_F6
1  x  grp     26   Bspl  x  grp     20    PCA      --      -     --
2  x  grp     26   Bspl  x  grp     20    PCA      --      -     --
3     --      -     --  x  grp     20    PCA      --      -     --
4  x  grp     26   Bspl  x  grp     20    PCA      --      -     --
5  x  grp     26   Bspl      --      -     --      --      -     --
  F7 D_F7 Dim_F7 Bas_F7  Kern    Q2
1     --      -     -- mat32 0.964
2     --      -     -- mat32 0.963
3     --      -     -- mat32 0.961
4     --      -     -- mat32 0.957
5     --      -     -- mat32 0.952
```

8. Closing discussion

This article introduced the R package **funGp**, which allows the treatment of regression problems involving multiple scalar and/or functional inputs, and a scalar output, through the fairly general GP model. In addition to being one of the few packages providing functional-input regression under this type of model, **funGp** features the ability to automatically set the structural parameters of the model for prediction performance. So far, this is a distinctive characteristic rarely found in current open-source regression software. The package is also equipped with multiple plotting functions that enable the quick inspection of the out-

puts delivered by its main functions: `fgpm()`, `fgpm_factory()`, `predict()` and `simulate()`. Moreover, parallelization has been incorporated as an efficiency booster for the procedures of model construction and structural optimization. The robustness and stability of **funGp** has been validated on numerous analytic examples and on the real life early warning application.

We envisage the extension of the package in various directions, including the addition of complementary levels of the structural parameters already available (e.g., other projection methods, distance functions and kernel functions), incorporation of additional structural parameters (e.g., type of transformation function for the output) and implementation of alternative structural optimization methods (e.g., brute-force search and Genetic Algorithms). Our code implementations were made with these potential extensions in mind and thus, the package is architectured under the principle of modularity for easy extension. Further potential extensions, both in terms of methodology and implementation, are possible. It would be valuable to allow for functional outputs and for multi-dimensional functional inputs, for instance two-dimensional maps numerically given by matrices. For multi-dimensional functional inputs, tensor product splines may be considered. As discussed in Section 3, an extension of the **funGp** package from simple to universal Kriging could be investigated, hence allowing to include a trend in the GP model. Note also that the automatic structural configuration procedure of Section 5 currently aims at returning a "best" structural configuration. Since close-to-optimal predictive performances could be achieved by very different structural configurations, an interesting extension of the **funGp** package could be to carry out Bayesian model averaging of multiple structural configurations, in the spirit of Zhang and Taflanidis (2019). Another extension perspective of the **funGp** package is to allow for covariance computations for functional inputs based on L^2 norms applied directly to the functions rather than their projections, and weighted by functional correlation lengths. This would provide new distances, sharing commonalities with (6), but where $\theta_f^{(k)}$ would be a function. The **funGp** package is available from GitHub (`https://github.com/djbetancourt-gh/funGp`) and CRAN (`https://cran.r-project.org/package=funGp`). We encourage the community to make contributions in the proposed topics or any other found relevant.

Acknowledgments

The **funGp** package was conceived within the RISCOPE project, funded by the French Agence Nationale de la Recherche (grant ANR-16-CE04-0011). We thank the ANR for its support. We are also grateful to Fabrice Gamboa (IMT, ANITI) for his helpful recommendations on GP modeling and to Juliette Garcia (ENAC) for her assistance on the stabilization of the Ant Colony algorithm for structural parameter optimization.

References

Bachoc F, Ammar K, Martinez JM (2016). "Improvement of Code Behavior in a Design of Experiments by Metamodeling." *Nuclear Science and Engineering*, **183**(3), 387–406.

Ben Salem M, Bachoc F, Roustant O, Gamboa F, Tomaso L (2019). "Gaussian Process-Based Dimension Reduction for Goal-Oriented Sequential Design." *SIAM/ASA Journal on Uncertainty Quantification*, **7**(4), 1369–1397.

Betancourt J (2020). *Functional-Input Metamodeling: An Application to Coastal Flood Early Warning*. Ph.D. thesis, Université Toulouse 3-Paul Sabatier.

Betancourt J, Bachoc F, Klein T, Gamboa (2020a). "Technical Report: Ant Colony Based Model Selection for Functional-Input Gaussian Process Regression. Ref. D3.b (WP3.2), *RISCOPE Project*." https://hal.archives-ouvertes.fr/hal-02532713. Hal-02532713.

Betancourt J, Bachoc F, Klein T, Idier D, Pedreros R, Rohmer J (2020b). "Gaussian Process Metamodeling of Functional-Input Code for Coastal Flood Hazard Assessment." *Reliability Engineering & System Safety*, p. 106870.

Betancourt J, Bachoc F, Klein T, Rohmer J (2020c). ***funGp**: Gaussian Process Models for Scalar and Functional Inputs*. R package version 0.3.0, URL https://cran.r-project.org/package=funGp.

Blum C (2005). "Ant Colony Optimization: Introduction and Recent Trends." *Physics of Life Reviews*, **2**(4), 353–373.

Bonabeau E, Dorigo M, Theraulaz G (1999). *Swarm Intelligence: From Natural to Artificial Systems*. 1. Oxford University Press.

Brockhaus S, Ruegamer D (2018). ***FDboost**: Boosting Functional Regression Models*. URL https://www.jstatsoft.org/article/view/v094i10.

Chen T, Dai B, Wang R, Liu D (2014). "Gaussian-Process-Based Real-Time Ground Segmentation for Autonomous Land Vehicles." *Journal of Intelligent & Robotic Systems*, **76**(3-4), 563–582.

Cheng Y, Shi JQ (2016). ***flars**: Functional LARS*. R package version 1.0, URL https://CRAN.R-project.org/package=flars.

Cressie N (1990). "The Origins of Kriging." *Mathematical Geology*, **22**(3), 239–252.

De Boor C (1978). *A Practical Guide to Splines*. Springer-Verlag.

De G Matthews AG, Van Der Wilk M, Nickson T, Fujii K, Boukouvalas A, León-Villagrá P, Ghahramani Z, Hensman J (2017). "**GPflow**: A Gaussian Process Library Using **TensorFlow**." *The Journal of Machine Learning Research*, **18**(1), 1299–1304.

Deville Y, Ginsbourger D, Roustant O (2020). ***kergp**: Gaussian Process Laboratory*. R package version 0.5.5, URL https://CRAN.R-project.org/package=kergp.

Dorigo M, Gambardella LM (1997). "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem." *IEEE Transactions on Evolutionary Computation*, **1**(1), 53–66.

Drucker H, Burges CJ, Kaufman L, Smola AJ, Vapnik V (1997). "Support Vector Regression Machines." In *Advances in Neural Information Processing Systems*, pp. 155–161.

Dubrule O (1983). "Cross Validation of Kriging in a Unique Neighborhood." *Journal of the International Association for Mathematical Geology*, **15**(6), 687–699.

Febrero-Bande M, Oviedo de la Fuente M (2012). "Statistical Computing in Functional Data Analysis: The R Package **fda.usc**." *Journal of Statistical Software*, **51**(4), 1–28. URL http://www.jstatsoft.org/v51/i04/.

Ferraty F, Vieu P (2006). "Reference Manual for Implementing NonParametric Functional Data Analysis (NPFDA)." *Companion Manual of the Book: Nonparametric Functional Data Analysis, Theory and Practice, Springer-Verlag, New York*.

Fonseca DJ, Navaresse DO, Moynihan GP (2003). "Simulation Metamodeling Through Artificial Neural Networks." *Engineering Applications of Artificial Intelligence*, **16**(3), 177–183.

Forrester A, Sobester A, Keane A (2008). *Engineering Design via Surrogate Modelling: A Practical Guide*. John Wiley & Sons.

Friedman JH (1991). "Multivariate Adaptive Regression Splines." *The Annals of Statistics*, **19**(1), 1–67.

Gajardo A, Bhattacharjee S, Carroll C, Chen Y, Dai X, Fan J, Hadjipantelis PZ, Han K, Ji H, Zhu C, Müller HG, Wang JL (2021). ***fdapace**: Functional Data Analysis and Empirical Dynamics*. R package version 0.5.8, URL https://CRAN.R-project.org/package=fdapace.

García-Portugués E, Álvarez Liébana J (2021). ***goffda**: Goodness-of-Fit Tests for Functional Data*. R package version 0.1.0, URL https://CRAN.R-project.org/package=goffda.

Goldsmith J, Scheipl F, Huang L, Wrobel J, Di C, Gellar J, Harezlak J, McLean MW, Swihart B, Xiao L, Crainiceanu C, Reiss PT (2020). ***refund**: Regression with Functional Data*. R package version 0.1-22, URL https://CRAN.R-project.org/package=refund.

GPy (2012). "**GPy**: A Gaussian Process Framework in Python." http://github.com/SheffieldML/GPy.

Gu M, Palomo J, Berger J (2020). ***RobustGaSP**: Robust Gaussian Stochastic Process Emulation*. R package version 0.6.1, URL https://CRAN.R-project.org/package=RobustGaSP.

Idier D, Aurouet A, Bachoc F, Bailly A, Betancourt J, Gamboa F, Klein T, López-Lopera AF, Pedreros R, Rohmer J, Thibault A (2021). "A User-Oriented Local Coastal Flooding Early Warning System Using Metamodelling Techniques." *Journal of Marine Science and Engineering*, **9**(11), 1191.

Iooss B, Marrel A (2019). "Advanced Methodology for Uncertainty Propagation in Computer Experiments with Large Number of Inputs." *Nuclear Technology*, **205**(12), 1588–1606.

Jolliffe I (2002). *Principal Component Analysis*. Springer-Verlag. 2nd edition.

Lataniotis C, Marelli S, Sudret B (2020). "Extending classical surrogate modelling to high-dimensions through supervised dimensionality reduction: a data-driven approach." *arxiv:1812.06309*.

Lee Y, Park JS (2017). "Model Selection Algorithm in Gaussian Process Regression for Computer Experiments." *Communications for Statistical Applications and Methods*, **24**(4), 383–396.

Li Y, Soleimani H, Zohal M (2019). "An Improved Ant Colony Optimization Algorithm for the Multi-Depot Green Vehicle Routing Problem with Multiple Objectives." *Journal of Cleaner Production*, **227**, 1161–1172.

Liu X, Guillas S (2017). "Dimension Reduction for Gaussian Process Emulation: An Application to the Influence of Bathymetry on Tsunami Heights." *SIAM/ASA Journal on Uncertainty Quantification*, **5**(1), 787–812.

Markussen B (2019). ***fdaMixed***: *Functional Data Analysis in a Mixed Model Framework*. R package version 0.6, URL https://CRAN.R-project.org/package=fdaMixed.

Marrel A, Iooss B, Van Dorpe F, Volkova E (2008). "An Efficient Methodology for Modeling Complex Computer codes with Gaussian Processes." *Computational Statistics & Data Analysis*, **52**(10), 4731–4744.

Mori H, Kurata E (2008). "Application of Gaussian Process to Wind Speed Forecasting for Wind Power Generation." In *2008 IEEE International Conference on Sustainable Energy Technologies*, pp. 956–959. IEEE.

Muehlenstaedt T, Fruth J, Roustant O (2017). "Computer Experiments with Functional Inputs and Scalar Outputs by a Norm-Based Approach." *Statistics and Computing*, **27**(4), 1083–1097.

Nanty S, Helbert C, Marrel A, Pérot N, Prieur C (2016). "Sampling, Metamodeling, and Sensitivity Analysis of Numerical Simulators with Functional Stochastic Inputs." *SIAM/ASA Journal on Uncertainty Quantification*, **4**(1), 636–659.

Neufeld A, Heggeseth B (2019). ***splinetree***: *Longitudinal Regression Trees and Forests*. R package version 0.2.0, URL https://CRAN.R-project.org/package=splinetree.

Nilsson J, de Jong S, Smilde AK (1997). "Multiway Calibration in 3D QSAR." *Journal of Chemometrics: A Journal of the Chemometrics Society*, **11**(6), 511–524.

Oakley J, O'Hagan A (2002). "Bayesian Inference for the Uncertainty Distribution of Computer Model Outputs." *Biometrika*, **89**(4), 769–784.

Østergård T, Jensen RL, Maagaard SE (2018). "A Comparison of six Metamodeling Techniques Applied to Building Performance Simulations." *Applied Energy*, **211**, 89–103.

Papaioannou I, Ehre M, Straub D (2019). "PLS-Based Adaptation for Efficient PCE Representation in High Dimensions." *Journal of Computational Physics*, **387**, 186–204.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). "**Scikit-learn**: Machine Learning in Python." *Journal of Machine Learning Research*, **12**, 2825–2830.

Ramsay JO, Graves S, Hooker G (2022). ***fda***: *Functional Data Analysis*. R package version 6.0.3, URL https://CRAN.R-project.org/package=fda.

Ramsay JO, Silverman BW (2007). *Applied Functional Data Analysis: Methods and Case Studies*. Springer-Verlag.

Rasmussen CE, Nickisch H (2010). "Gaussian Processes for Machine Learning (**GPML**) Toolbox." *The Journal of Machine Learning Research*, **11**, 3011–3015.

Rasmussen CE, Williams CKI (2006). *Gaussian Processes for Machine Learning*, volume 2. MIT Press Cambridge, MA.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Rohmer J, Idier D (2012). "A Meta-Modelling Strategy to Identify the Critical Offshore Conditions for Coastal Flooding." *Natural Hazards & Earth System Sciences*, **12**(9).

Roustant O, Ginsbourger D, Deville Y (2012). "**DiceKriging**, **DiceOptim**: Two R Packages for the Analysis of Computer Experiments by Kriging-Based Metamodeling and Optimization." *Journal of Statistical Software*, **51**(1), 1–55. URL http://www.jstatsoft.org/v51/i01/.

Sacks J, Welch WJ, Mitchell TJ, Wynn HP (1989). "Design and Analysis of Computer Experiments." *Statistical Science*, **4**, 409–423.

Saltelli A, Tarantola S, Campolongo F, Ratto M (2004). *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*, volume 1. John Wiley & Sons.

Santner TJ, Williams BJ, Notz WI, Williams BJ (2003). *The Design and Analysis of Computer Experiments*, volume 1. Springer-Verlag.

Savitsky TD (2016). "Bayesian Nonparametric Mixture Estimation for Time-Indexed Functional Data in R." *Journal of Statistical Software*, **72**(2), 1–34. doi:10.18637/jss.v072.i02.

Shi JQ, Cheng Y (2014). ***GPFDA**: Apply Gaussian Process in Functional Data Analysis*. R package version 2.2, URL https://CRAN.R-project.org/package=GPFDA.

Simpson TW, Poplinski J, Koch PN, Allen JK (2001). "Metamodels for Computer-Based Engineering Design: Survey and Recommendations." *Engineering with Computers*, **17**(2), 129–150.

Singh SS, Singh K, Kumar A, Biswas B (2020). "ACO-IM: Maximizing Influence in Social Networks Using Ant Colony Optimization." *Soft Computing*, **24**(13), 10181–10203.

Tsiliki G, Munteanu C, Seoane J, Fernandez-Lozano C, Sarimveis H, Willighagen E (2015). "**RRegrs**: An R Package for Computer-Aided Model Selection with Multiple Regression Models." *Journal of Cheminformatics*, **7**(1), 46. URL http://dx.doi.org/10.1186/s13321-015-0094-2.

Vanhatalo J, Riihimäki J, Hartikainen J, Jylänki P, Tolvanen V, Vehtari A (2012). "Bayesian Modeling with Gaussian Processes Using the **GPstuff** Toolbox." *arXiv preprint arXiv:1206.5754*.

Yuan J, Nian V (2018). "Ship Energy Consumption Prediction with Gaussian Process Metamodel." *Energy Procedia*, **152**, 655–660.

Zhang J, Taflanidis A (2019). "Bayesian Model Averaging for Kriging Regression Structure Selection." *Probabilistic Engineering Mechanics*, **56**, 58–70.

A. funGp at a glance

The **funGp** package allows the easy construction of GP regression models with special treatment of functional inputs, an aspect scarcely treated by other R packages. The package is constituted by two major modules which are described in the paragraphs that follow.

◇ **Regression:** this module is responsible for the core functionalities of the package: (i) creation and diagnostic of GP regression models, (ii) prediction at unobserved input coordinates, (iii) conditional simulation, and (iv) model updating. These procedures are implemented in the five functions listed in Table 2. All the listed functions are set up with default arguments that allow an effortless first interaction with the package. However, high degree of customization is enabled to let the users get greater control over the outputs and extend the scope of the analysis as they become familiar with the overall functioning of the package. As an example, the `fgpm()` function allows the customization of the so-called structural parameters and the definition of custom hyperparameters, among other features.

Function	Type	Description
`fgpm()`	function	Creation of GP regression models. Outputs an object of S4 class `"fgpm"` representing the fitted model
`predict()`	method	Prediction at new input points. Delivers the estimated output values and corresponding uncertainty at specified points
`simulate()`	method	Conditional simulation. Delivers the simulated output values and corresponding uncertainty at specified points
`update()`	method	Efficient model update. Allows to add, delete or substitute data, and also to replace or re-estimate hyperparameters
`plot()`	method	Diagnostic plot for GP model; receives an `fgpm` object as main argument

Table 2: Base **funGp** functionalities related to the class `"fgpm"`.

◇ **Structural optimization:** the set of functions listed in Table 2 could constitute by themselves a regression package. However, **funGp** goes further and incorporates the automatic calibration of structural parameters of the model, a functionality widely neglected by other regression tools. Structural optimization enables the efficient exploration of the space of possible model configurations, seeking for optimal predictive performance. Diagnostic plots are provided for a quick evaluation of the selected model and the evolution of the optimization. The main procedures of this module are implemented in the three functions listed in Table 3.

Function	Type	Description
`fgpm_factory()`	function	Structural optimization of GP models. Outputs an object of S4 class `"Xfgpm"` containing the optimized model along with records from the optimization process
`plot()`	method	`which = "diag"`: information on the absolute and relative quality of the selected model
`plot()`	method	`which = "evol"`: verification plot showing the evolution in the quality of the explored structural configurations along the optimization
`summary()`	method	Lists the evaluated structural configurations with their prediction performances

Table 3: Complementary **funGp** functionnalities related to the class `"Xfgpm"` enabling structural optimization.

B. Manual prediction

As explained in Section 3.2, GP predictions are accessible in **funGp** through the `predict()` method. However, as a verification/inspection procedure, it is also possible to recreate the different data structures necessary for making the predictions manually based on the equations (2) and (3) of Gaussian conditioning. The recovery of these data structures is explained below, departing from an hypothetical model `m1` created with `fgpm()`.

- Vector of observed output values $\boldsymbol{y}$: this vector is stored at the `@sOut` slot of the `fgpm` object associated with the model.

```
R> y <- m1@sOut
```

- Unconditional $n \times n_*$ training-prediction cross-covariance matrix $C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}_*, \boldsymbol{F}_*))$ and $n_* \times n_*$ prediction auto-covariance matrix $C((\boldsymbol{X}_*, \boldsymbol{F}_*), (\boldsymbol{X}_*, \boldsymbol{F}_*))$: these matrices can be obtained from a *full prediction* generated by `predict()`, as shown below.

```
R> preds <- predict(m1, sIn.pr = sIn.pr, fIn.pr = fIn.pr, detail = "full")
R> K.tp <- preds$K.tp
R> K.pp <- preds$K.pp
```

- Unconditional $n \times n$ training auto-covariance matrix $C((\boldsymbol{X}, \boldsymbol{F}), (\boldsymbol{X}, \boldsymbol{F}))$: this matrix can be regenerated from the lower triangular matrix of its LU decomposition, stored at the `@preMats$L` slot of the model.

```
R> K.tt <- tcrossprod(m1@preMats$L)
```

Once the `y`, `K.tp`, `K.pp` and `K.tt` data structures have been recovered, the predicted output values and corresponding confidence intervals can be reproduced based on Gaussian conditioning as follows.

```
R> ## predictive mean and variance
R> y.pr <- t(K.tp) %*% solve(K.tt) %*% y
```

```
R> v.pr <- diag(K.pp - t(K.tp) %*% solve(K.tt) %*% K.tp)
R> ## limits of 95% confidence intervals
R> ll <- y.pr - 1.96 * sqrt(v.pr)
R> ul <- y.pr + 1.96 * sqrt(v.pr)
```

We remark that this way of computing GP predictions is computationally inefficient, and is only recommended for purposes of verification and examination of the involved data structures. Predictions and simulations performed through the `predict()` and `simulate()` functions in **funGp** are highly optimized for computational efficiency.

Affiliation:

José Betancourt
Institut de Mathématiques de Toulouse (IMT), Université de Toulouse
École nationale de l'aviation civile (ENAC)
118 Route de Narbonne
31062 Toulouse Cedex 9, Françe
E-mail: djbetancourt@uninorte.edu.co
URL: https://www.linkedin.com/in/djbetancourt

François Bachoc
IMT, Université de Toulouse
E-mail: francois.bachoc@math.univ-toulouse.fr
URL: https://www.math.univ-toulouse.fr/~fbachoc

Thierry Klein
ENAC - IMT, Université de Toulouse
E-mail: thierry.klein@math.univ-toulouse.fr
URL: https://perso.math.univ-toulouse.fr/klein

Déborah Idier
Bureau de Recherches Géologiques et Minières (BRGM)
3, av. Claude Guillemin
45060 Orléans Cedex 2, Françe
E-mail: D.Idier@brgm.fr
URL: https://www.researchgate.net/profile/Deborah_Idier2

Jérémy Rohmer
BRGM
E-mail: J.Rohmer@brgm.fr
URL: https://www.researchgate.net/profile/Jeremy_Rohmer

Yves Deville
Alpestat
E-mail: deville.yves@alpestat.com
URL: https://www.alpestat.com