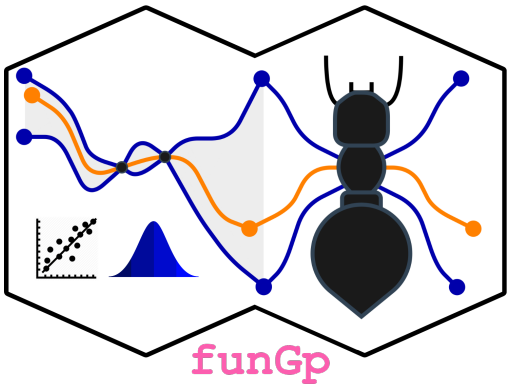
Gaussian Process Regression for Scalar and Functional Inputs with funGp - The in-depth tour

José Daniel Betancourt, François Bachoc, Thierry Klein



Gaussian Process Regression for Scalar and Functional Inputs with funGp

The in-depth tour

This is a comprehensive guide to creating and manipulating Gaussian process regression models using the R package **funGp**. It illustrates through examples, the usage of every function in the package and each example is accompanied by a code snippet to shorten the learning curve through direct usage of the functions.

Authors: José Betancourt, François Bachoc, Thierry Klein.
Contributors: Déborah Idier, Jérémy Rhomer.

This manual is for **funGp**, version 0.1.0 (2020), downloadable from CRAN and GitHub.

Recommended citation: Betancourt, J., Bachoc, F., Klein, T. (2020). R Package Manual: Gaussian Process Regression for Scalar and Functional Inputs with funGp - The in-depth tour. *RISCOPE project*.

funGp was first developed in the frame of the RISCOPE research project, funded by the French Agence Nationale de la Recherche (ANR) for the period 2017-2021 (ANR project No. 16CE04-0011, RISCOPE.fr), and certified by SAFE Cluster.

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 assumptions 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 [1, 2] 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.

Contents

In-code notation

`n.tot`	Number of points used for prediction
`n.tr`	Number of points using for learning of hyperparameters
`n.pr`	Number of prediction points
`n.sm`	Number of simulation points
`ds`	Number of scalar inputs
`df`	Number of functional inputs
`k`	Array of dimensions for the *df* functional inputs
`p`	Array of projection dimensions for the functional inputs
`K.tt`	Training auto-covariance matrix
`K.pp`	prediction auto-covariance matrix
`K.tp`	Training-prediction cross-covariance matrix
`L`	Lower diagonal matrix of a Cholesky decomposition

1 Base functionalities

This section starts from the bottom with the fundamental tasks implemented in **funGp**. Those are: (i) creation of regression models, (ii) prediction of the output at unobserved input points, (iii) simulation of trajectories from the underlying Gaussian process linked to any funGp model, and (iv) updating an existing model.

The workflow for each of the four functionalities listed above is illustrated through a follow-along example based on the analytic black-box function

$$\begin{aligned}\mathcal{G}_1: \quad & [0,1]^2 \times \mathcal{F}^2 \to \mathbb{R}, \\ & (\boldsymbol{x}, \boldsymbol{f}) \mapsto x^{(1)} + 2x^{(2)} + 4\int_0^1 t f^{(1)}(t)\, dt + \int_0^1 f^{(2)}(t)\, dt,\end{aligned}$$

with $\boldsymbol{x} = \left(x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}, x^{(5)}\right)$ the scalar inputs, $\boldsymbol{f} = \left(f^{(1)}, f^{(2)}\right)$ the functional inputs, and $\mathcal{F}$ the set of continuous functions from $[0, 1]$ to $\mathbb{R}$. This function corresponds to the first analytic example presented in [3], and is accessible in **funGp** through the black-box function fgp_BB3.

All code snippets are copy/paste-able directly to R.

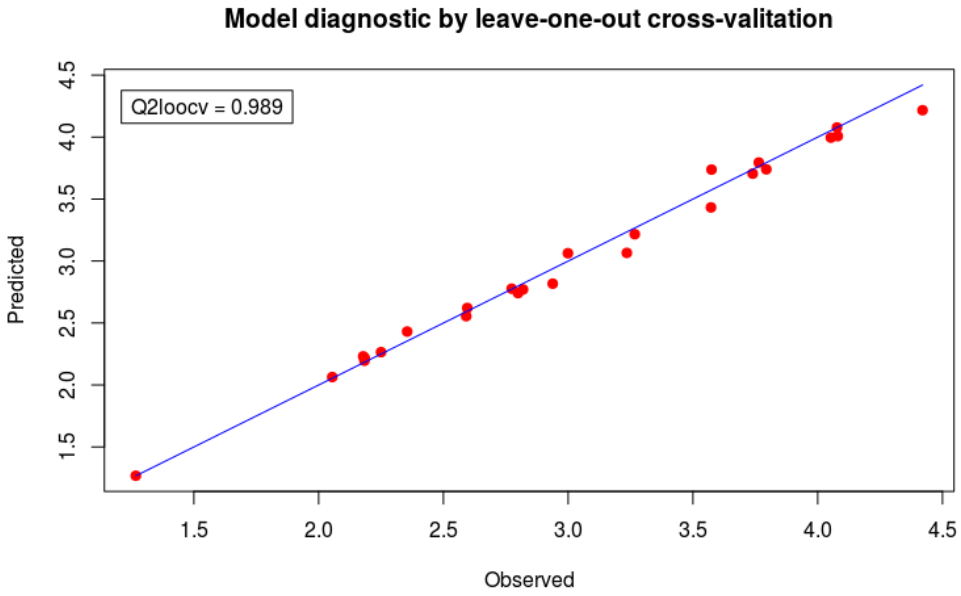
1.1 Create a funGp model

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. The output should be provided as an **array** or single-colum **matrix**. In the case of the inputs, each row of a **matrix** must correspond to an input point. Here, we will use synthetic data based on the analytic case defined in the introductory paragraph of this section, which involves two scalar inputs and two functional inputs. To generate the input data, we took the scalar input points from a factorial design over $[0, 1]$. For the functional inputs we assumed that those were measured at 10 and 22 time instants. This, just to emphasize the fact that functional inputs with heterogeneous discretization are valid funGp inputs. Just to have some data to work with, we sampled all the values of each function randomly from $U(0, 1)$. We also picked an arbitrary number of 25 training points.

```
# generating input data for training
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))

# generating output data for training
sOut <- fgp_BB3(sIn, fIn, n.tr)

# creating a funGp model
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut)

R output:
** Presampling...
** Optimising...
final  value 2.841058 # loglikelihood value
converged
```

The output of the `fgpm` function is an object of class `fgpm`. A calibration plot based on the Leave-one-out (LOO) predictions.

```
# plotting the model
plotLOO(m1)
```



For a design with $n.tr$ points, LOO consists of removing one observation from the design at a time, each time training the model using the remaining $n.tr - 1$ points and computing the prediction at the ignored point. In its basic version, LOO results expensive as it requires training $n.tr$ models using almost all the data each time. For Gaussian processes, the LOO predictions are often approximated based on the virtual LOO formulas [4, 5], which require a single model training.

The model diagnostic plot also displays a measure of the external prediction capability of the model. It corresponds to the LOO cross-validated squared correlation coefficient Q^2_{loocv}, defined as:

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

with $(y_i)_{i=1,\ldots,n.tr}$ the vector of observed output values, $\bar{y}$ the average of that vector and $\hat{y}_{i,-i}$ the LOO prediction of y_i.

Main features of the model are printed when calling the `show` function on the model:

```
# printing the model
m1 # equivalent to show(m1)

R output:
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`** is related to the possibility of requesting DR[1] for the functional inputs. DR allows to project a functional input of dimension k_i onto a space of lower dimension p_i while preserving the main statistical or geometric properties of the variable [6, 7]. This process often leads to $p_i << k_i$, which improves the tractability and processing speed of the model. By default, the `fgpm` function sets $p_i = 3$ for all the functional inputs. The user is allowed to pick a custom projection dimension for each input and also to do not project some of them. Different projection methods (basis families) are also available. The projection method used for each input is indicated under the field **`Basis`**. The manipulation of the projection dimension and projection method are discussed in more detail in Section 2.2 and Section 2.3, respectively.

1.2 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 1.1. 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.

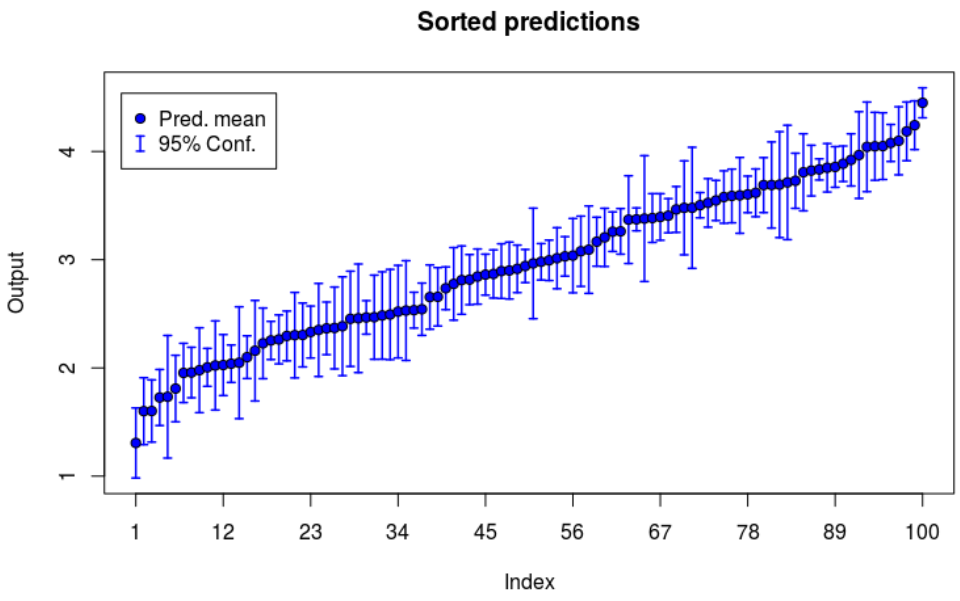
For the example, we generated the input points in a similar way as for training, i.e., the scalar inputs from a factorial design over $[0, 1]$ and the functional values randomly from $U(0, 1)$.

[1] DR: dimension reduction.

```
# building the model
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB3(sIn, fIn, n.tr)
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut)

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

# making predictions
m1.preds <- predict(m1, sIn.pr = sIn.pr, fIn.pr = fIn.pr)

# checking content of the list
summary(m1.preds)

R output:
        Length Class  Mode
mean    100    -none- numeric
sd      100    -none- numeric
lower95 100    -none- numeric
upper95 100    -none- numeric
```

The output of `predict` is a **`list`** containing the estimated mean and standard deviation, along with the lower and upper limits of the 95% confidence intervals for the output at the prediction points. In practice, the estimated mean of a Gaussian process 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. Predictions of a `funGp` model can be easily plotted by calling the `plotPreds` function on the **`list`** returned by `predict`. Note that the model must also be sent in the function call.

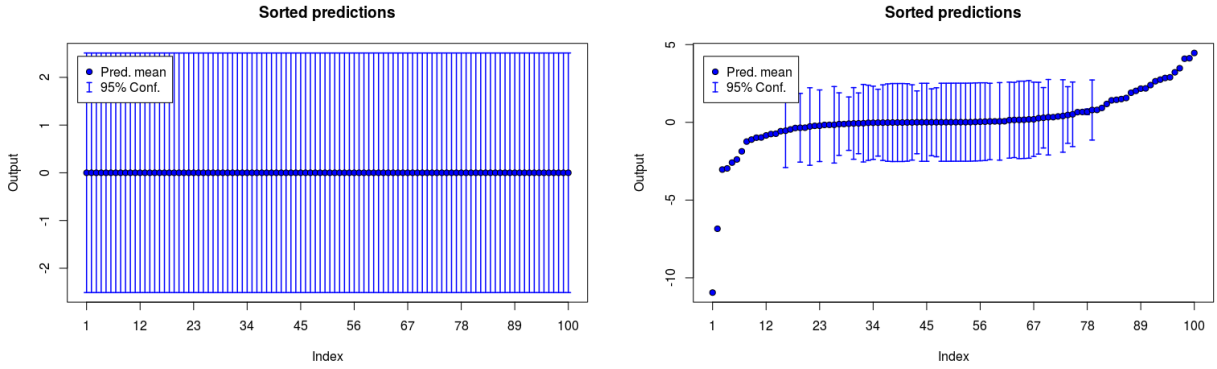
```
# plotting predictions
plotPreds(m1, preds = m1.preds)
```



With functional inputs, the simple Out-vs-In scatter plots are no longer an option. Thus, `plotPreds` displays the increasingly sorted mean and corresponding confidence intervals instead. This plot can be used as a diagnostic tool for identifying potential problems related to the hyperparameters optimization, for instance:

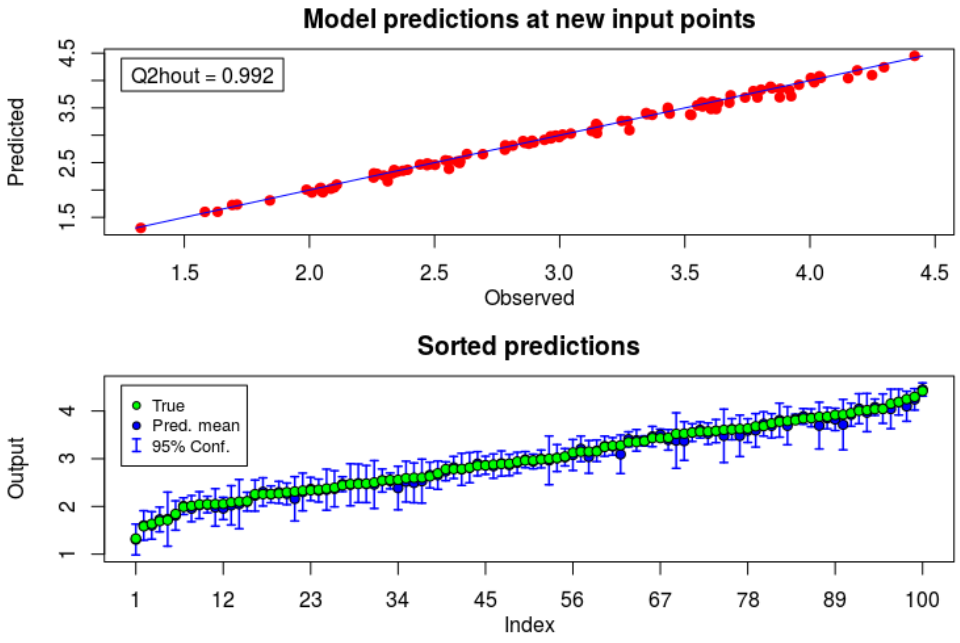
- Excessively wide confidence intervals could indicate a far-from-optimal hyperparameters' estimation, especially if it happens for all or a large number of prediction points;
- When a prediction point is included in the training set, the model interpolates the output and no confidence interval is displayed for that point. In any other case, missing confidence intervals may be indicative of far-from-optimal hyperparameters' estimation.

Figures illustrating the two aforementioned potential scenarios are displayed below.



The `plotPreds` function can also be used to compare predictions against true output values. In that case, an observed-vs-predicted calibration plot will be added on top of the sorted-output plot shown before.

```
# validating against true output
sOut.pr <- fgp_BB3(sIn.pr, fIn.pr, n.pr)
plotPreds(m1, m1.preds, sOut.pr)
```



The calibration plot made by `plotPreds` will display the predictive squared correlation coefficient Q^2_{hout}, which corresponds to the classical coefficient of determination R^2 for a test sample, i.e., for prediction residuals [8]. On the other hand, the ordering in the sorted-output plot will be lead by the true output vector instead of the predicted mean vector. This way of sorting is convenient for comparing results of different models fitting the same data. Either the calibration plot or the sorted-output plot can be displayed alone by specifying the argument **`sortp = FALSE`** or **`calib = FALSE`**, respectively, when calling `plotPreds`.

Note: 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 training-prediction cross-covariance matrix `K.tp` and the prediction auto-covariance matrix `K.pp`. To make full predictions, it suffices to set `detail = "full"` when calling `predict`. The behavior of `plotPreds` is not affected by this selection.

```
# making full predictions
m1.preds_f <- predict(m1, sIn.pr = sIn.pr, fIn.pr = fIn.pr, detail = "full")

# checking content of the list
summary(m1.preds_f)

R output:
        Length Class  Mode
mean       100  -none- numeric
sd         100  -none- numeric
K.tp      2500  -none- numeric
K.pp     10000  -none- numeric
lower95    100  -none- numeric
upper95    100  -none- numeric
```

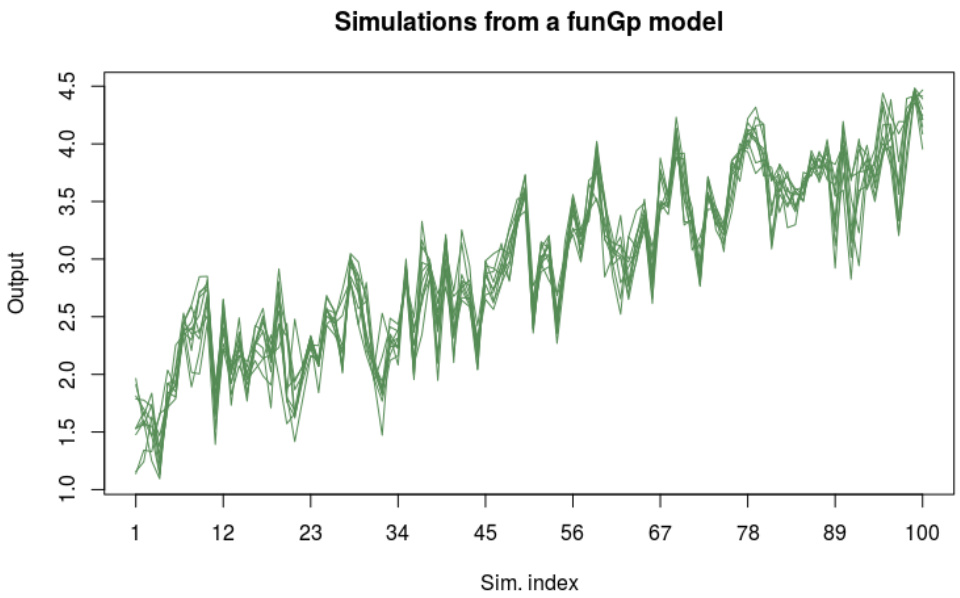
1.3 Simulate from a `funGp` model

Simulations in **`funGp`** are requested through the `simulate` function, in a similar way to predictions. 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. Each row of an input **`matrix`** will be interpreted as a point at which to provide simulations. By default, `simulate` will perform so-called *light simulations*, returning a $n.rep \times n.sm$ **`matrix`**, with $n.rep$ the number of replications to produce at each input point and $n.sm$ the number of input points. For the example we took the scalar inputs from a factorial design over [0,1] and the functional values randomly from $U(0, 1)$.

```
# building the model
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB3(sIn, fIn, n.tr)
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut)

# generating input data for simulation
n.sm <- 100
sIn.sm <- as.matrix(expand.grid(x1 = seq(0,1,length = sqrt(n.sm)), x2 = seq(0,1,length = sqrt(n.sm))))
fIn.sm <- list(f1 = matrix(runif(n.sm*10), ncol = 10), matrix(runif(n.sm*22), ncol = 22))

# making light simulations
m1.sims_l <- simulate(m1, nsim = 10, sIn.sm = sIn.sm, fIn.sm = fIn.sm)
```

Simulations in **funGp** are plotted by the plotSims function. In contrast to prediction plots, simulation plots do not have the output sorted in increasing order, but instead, the simulation index corresponding to the input coordinates specified by the user is set in the abscissa.

```
# plotting light simulations
plotSims(m1, m1.sims_l)
```

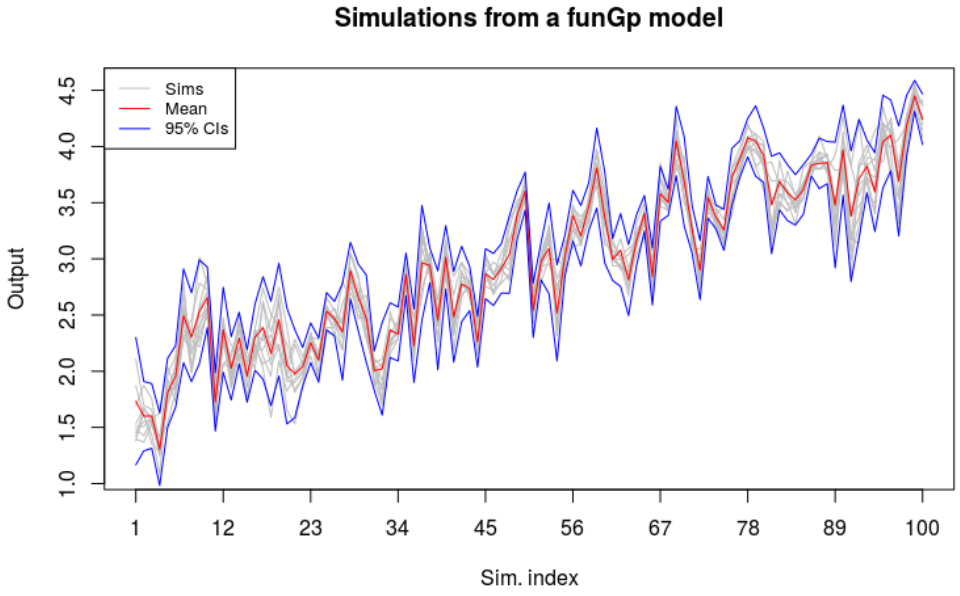


If requested, simulate will return a **list** containing the simulated output, predicted mean, standard deviation and limits of the 95% confidence intervals at the specified input coordinates. This corresponds to a *full simulation*, available through the option `detail = "full"`.

```
# making full simulations
m1.sims_f <- simulate(m1, nsim = 10, sIn.sm = sIn.sm, fIn.sm = fIn.sm, detail = "full")

# checking content of the list
summary(m1.sims_f)

R output:
        Length Class  Mode
sims    1000   -none- numeric
mean     100   -none- numeric
sd       100   -none- numeric
lower95  100   -none- numeric
upper95  100   -none- numeric
```

Full simulations can also be plotted using the plotSims function. By default, the plot of full simulations will include the predicted mean and limits of the confidence intervals.

```
# plotting full simulations in full mode
plotSims(m1, m1.sims_f)
```



A light plot without the mean and confidence intervals is also available for full simulations by setting `detail = "light"` when calling `plotSims`.

1.4 Update a `funGp` model

As simple as it might appear, the `update` function allows to perform nine different updating tasks on a `funGp` model:

- Operations over the **`@sIn`**, **`@fIn`** and **`@sOut`** slots
    1. Deletion of data points
    2. Substitution of data points
    3. Addition of data points
- Operations over the **`@kern@varHyp`**, **`@kern@s_lsHyps`** and **`@kern@s_lsHyps`** slots
    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
    9. Re-estimation of the vector of functional length-scale hyperparameters

There are many reasons why you might want to modify an existing model; new observations became available, some of those used for training became obsolete, transcription or typing errors were found in the training data, you want to experiment with different values of the hyperparameters, just to mention some. In most cases, part of the work done during the construction of the original model can be exploited to make the updating process much faster than building a new model from zero. The request of the different updating tasks is illustrated in the code snippets below. If you have not built a model yet using the code

provided in previous sections, you can use the following one to obtain a model on which to perform the update tasks of the upcoming examples.

```
# building the model
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB3(sIn, fIn, n.tr)
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut)
```

- **Deletion and addition of data points**

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

R output:
* Complete tasks:
    - data deletion

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

R output:
* Complete tasks:
    - data addition
```

- **substitution of data points**

```
# generating substituting input data for updating
n.sb <- 2
sIn.sb <- matrix(runif(n.sb * m1@ds), nrow = n.sb)
fIn.sb <- list(f1 = matrix(runif(n.sb*10), ncol = 10), f2 = matrix(runif(n.sb*22), ncol = 22))

# generating substituting output data for updating
sOut.sb <- fgp_BB3(sIn.sb, fIn.sb, n.sb)

# generating indices for substitution
ind.sb <- sample(1:(m1@n.tot), n.sb)

# updating all, the scalar inputs, functional inputs and the output
m1up <- update(m1, sIn.sb = sIn.sb, fIn.sb = fIn.sb, sOut.sb = sOut.sb, ind.sb = ind.sb)

R output:
* Complete tasks:
    - data substitution
```

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

```
# substituting some data structures
m1up1 <- update(m1, sIn.sb = sIn.sb, ind.sb = ind.sb) # only the scalar inputs
m1up2 <- update(m1, sOut.sb = sOut.sb, ind.sb = ind.sb) # only the output
m1up3 <- update(m1, sIn.sb = sIn.sb, sOut.sb = sOut.sb, ind.sb = ind.sb) # the scalar inputs and the output

R output:
* Complete tasks:
    - data substitution
```

- **Substitution of hyperparameters**

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

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

R output:
* Complete tasks:
    - var substitution
    - scalar length-scale substitution
    - functional length-scale substitution
```

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

```
# updating the model
m1up <- update(m1, var.sb = var.sb) # only the variance
m1up <- update(m1, ls_f.sb = ls_f.sb) # only the functional length-scale parameters
m1up <- update(m1, var.sb = var.sb, ls_s.sb = ls_s.sb) # only the variance and the scalar ls. parameters
```

- **Re-estimation of hyperparameters**

```
# re-estimating the hyperparameters
m1up <- update(m1, var.re = TRUE) # only the variance
m1up <- update(m1, ls_s.re = TRUE) # only the scalar length-scale parameters
m1up <- update(m1, ls_s.re = TRUE, ls_f.re = TRUE) # all length-scale parameters
m1up <- update(m1, var.re = TRUE, ls_s.re = TRUE, ls_f.re = TRUE) # all hyperparameters

R output:
* Complete tasks:
    - var re-estimation
    - scalar length-scale re-estimation
    - functional length-scale re-estimation
```

It is possible to request multiple tasks from the different categories listed above in a single call to `update`. When doing so, it is convenient to keep in mind that tasks will be performed in the following order:

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

It is also good to remember that the following two combinations are unfeasible:

- Data points deletion and substitution;
- Substitution and re-estimation of the same hyperparameter.

2 Model customizations

There are multiple things we can do in order to improve the tractability and predictability of a `funGp` model. In this section we discuss the customization of the model through its so-called structural parameters. It refers to a set of categorical features such as the kernel function or the projection basis, whose levels could be alternated in order to generate different models departing from the same input-output data. Without going too deep into the technical details, this section explains how to start working on these features within **`funGp`** and the interested reader is referred to [1] for a formal and more detailed explanation of the underlying theory.

2.1 Kernel family

The selection of a suitable kernel function is something that naturally comes to mind when working with Gaussian process 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 when calling the fgpm function, through the parameter **kertype**. Valid values for this attribute are **"gauss"**, **"matern5_2"** and **"matern3_2"**. See for instance the example below with the Gaussian kernel.

```
# building the model
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB3(sIn, fIn, n.tr)
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, kerType = "gauss")
```

By default, fgpm uses the Matérn 5/2 function, which is a popular choice in the Machine Learning (ML) community.

2.2 Projection basis

In earlier sections of the manual we talked about DR[2], the process of reducing the dimension of your data structures in such a way and extent that the model becomes significantly more tractable and the loss in terms of predictability is negligible, if some. A common DR approach when dealing with functional inputs is to project each functional-input **matrix** onto a space of lower dimension. This method requires the construction of a set of basis vectors on which the original curves are projected. Those vectors (typically referred to as basis functions) may come from diverse families, including among the most popular ones the B-splines [9], PCA [10], PLS [11], wavelets [12] and kPCA [13]. The suitability of a given basis type might depend on the regression instance at hand. The B-splines and PCA bases are currently implemented in **funGp** for the projection of functional inputs. This option is accessible in the fgpm function through the parameter **f_basType**, which can be set to the values **"B-splines"** or **"PCA"**. When multiple functional inputs are provided, a custom basis can be selected for each of them, by passing an **array** with the selection for each input. If multiple functional inputs are provided, but a single **f_basType** value is specified, that selection is used for all the inputs. Both cases are illustrated below. By default, all functional inputs use a B-splines basis.

```
# generating input and output data
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB3(sIn, fIn, n.tr)

# building the model
# different basis for each functional input
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_basType = c("B-splines", "PCA"))

# same basis for both functional inputs
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_basType = "PCA")
```

[2]DR: dimension reduction.

2.3 Projection dimension

This parameter is highly influential in both, the predication quality and the tractability of the model. Ideally, one wants to set the projection dimension considerably lower than the original one, but not so low that significant prediction power is lost. In the `fgpm` function, you can specify the projection dimension for each input by setting the argument **`f_pdims`**. Valid inputs are all the integer numbers from 0 to the original dimension of the curves. The value 0 is used to request to not perform the projection of an input. If there are multiple functional inputs, an **`array`** can be provided instead of a single value in order to specify custom projection dimensions. If a single value is specified and multiple functional inputs are identified, the value is used as projection dimension for all the functional inputs. By default, all functional inputs are projected onto a space of dimension 3.

```
# generating input and output data
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB3(sIn, fIn, n.tr)

# building the model
# the first input not projected, the second one projected in dimension 7
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0, 7))

# both inputs projected in dimension 5
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = 5)
```

2.4 Distance for functions

Many regression models require the computation of the distance between the design points in order to determine which ones are the most influential in a given prediction. This is the case of Gaussian process 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 rule is simply to use one length-scale parameter per input, which yields the distance

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

with $\boldsymbol{x} = \left(x^{(1)}, \ldots, x^{(ds)}\right)$ and $\tilde{\boldsymbol{x}} = \left(\tilde{x}^{(1)}, \ldots, \tilde{x}^{(ds)}\right)$ two scalar input points, ds the number of scalar inputs in the model, $\|\cdot\|$ the L^2 norm for scalars (just the absolute value), and $\boldsymbol{\theta_s} = \left(\theta_s^{(1)}, \ldots, \theta_s^{(ds)}\right)$ the vector of length-scale parameters for the scalar inputs.

In an instance with functional inputs, the norm $\|\cdot\|$ needs to be replaced by a norm suitable for functions. Two options are currently implemented in `funGp`, both based on a projection of each functional inputs of the form

$$\Pi\left(f^{(k)}\right)(t) = \sum_{r=1}^{p_k} \alpha_r^{(k)} B_r^{(k)}(t), \tag{2}$$

with $f^{(k)}$ a curve of the k-th functional input, $B_r^{(k)}$ the r-th basis function used for its projection, $\alpha_r^{(k)}$ the corresponding projection coefficient, and p_k the projection dimension.

The first type of distance implemented for functions considers each curve as a whole and uses a single length-scale parameter per functional input. This distance is defined as

$$\left\|\Pi\left(\boldsymbol{f}\right) - \Pi\left(\tilde{\boldsymbol{f}}\right)\right\|_{D,\boldsymbol{\theta_f}} := \sqrt{\sum_{k=1}^{df} \frac{\int_{T_k} \left(\sum_{r=1}^{p_k} \left(\alpha_r^{(k)} - \tilde{\alpha}_r^{(k)}\right) B_r^{(k)}(t)\right)^2 dt}{\left(\theta_f^{(k)}\right)^2}}, \tag{3}$$

with $\boldsymbol{f} = \left(f^{(1)}, \ldots, f^{(df)}\right)$ and $\tilde{\boldsymbol{f}} = \left(\tilde{f}^{(1)}, \ldots, \tilde{f}^{(df)}\right)$ two functional input points, df the number of scalar inputs in the model, $T_k \subset \mathbb{R}$ the domain of $f^{(k)}$, and $\boldsymbol{\theta_f} = \left(\theta_f^{(1)}, \ldots, \theta_f^{(df)}\right)$ 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 corresponding to one functional input. **`funGp`** implements an efficient computation of (3), introduced in [3] and further studied in [1].

The second type of distance works only with the projection coefficients and disregards the basis functions. The distance is defined as

$$\left\|\Pi\left(\boldsymbol{f}\right) - \Pi\left(\tilde{\boldsymbol{f}}\right)\right\|_{S,\dot{\boldsymbol{\theta}}_{\boldsymbol{f}}} := \sqrt{\sum_{k=1}^{df} \sum_{r=1}^{p_k} \frac{\left(\alpha_r^{(k)} - \tilde{\alpha}_r^{(k)}\right)^2}{\left(\dot{\theta}_{f,r}^{(k)}\right)^2}}, \tag{4}$$

where $\dot{\boldsymbol{\theta}}_{\boldsymbol{f}} = (\dot{\theta}_{f,r}^{(k)})_{1 \le r \le p_k, 1 \le k \le df}$ denotes the vector of functional length-scale coefficients. Note that this distance uses one length-scale coefficient per projection term. This might enable a better modeling of the input-output relationship, but in turn it implies a larger number of decision variables involved in the learning process, which makes it a harder/longer task. This distance is identified in the package as **`L2_byindex`** since it involves a length-scale parameter per projection index. It corresponds to the most common approach nowadays, which is to perform the projection of the inputs and then use each projection coefficient as an individual scalar input of the model.

In the case that no projection is requested for some input, both distances (3) and (4) use the original function values instead of the projection coefficients, and the identity is used in (3) as the matrix of basis functions. Our aim is to keep this manual friendly with users not expert in statistics. Thus, we leave at this point the technical discussion on the distances, and we refer the interested user to [1], where this aspect is discussed formally and in more detail. Below, there are some examples on the selection of the distance through `fgpm`.

```
# generating input and output data
set.seed(100)
n.tr <- 25
sIn <- expand.grid(x1 = seq(0,1,length = sqrt(n.tr)), x2 = seq(0,1,length = sqrt(n.tr)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB3(sIn, fIn, n.tr)
```

```
# original dimensions
# f1: 10
# f2: 22

# building the model
# the first f. input using by-index distance and no projection -> 10 length-scale parameters
# the second f. input using by-group distance -> 1 length-scale parameter
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0,5), f_disType = c("L2_byindex", "L2_bygroup"))

# both f. inputs using by-group distance -> 2 length-scale parameters
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0,5), f_disType = "L2_bygroup")

# both f. inputs using by-index distance -> (10+5) = 15 length-scale parameters
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, f_pdims = c(0,5), f_disType = "L2_byindex")
```

3 Heuristic model selection

In the previous section, we covered the base functionalities of **funGp**. Now, we present a boosting feature that takes funGp models one step further: the **funGp** model factory.

3.1 Concept

The fgpm function, explored in Section 1.1, allows to specify through its arguments a number of characteristics of the model, oriented to make it adaptive to the particularities of the regression problem at hand. In Section 2 we catalogued those features under the name of structural parameters of the model, and we illustrated through examples the way of specifying the required configuration of them to the fgpm function. In its current version, **funGp** includes the kernel family, the projection basis, the projection dimension and the distance for functions as structural parameters modifiable by the user. But, which combination of structural parameters should you use? If you have strong evidence to think that some level of one of these features will perform better than the others, then you are good to go. Otherwise, it would be better to make some tests in order to make such a decision. As shown by us through a set of computer experiments in [1], the ideal model configuration might likely depend on the particular regression task. Through the fgpm_factory function we enable the user to conduct a smart exploration of the solution space composed of all the possible structural parameter configurations. Variable selection is embedded in the optimization through the definition of structural parameters related to the state of each scalar and functional input in the model (active or inactive).

At this point, **funGp** performs heuristic optimization of structural parameters (model selection) by means of the ant colony based algorithm introduced by us in [14] (find online ↗). For a set of *ds* scalar inputs and *df* functional inputs, the optimization problem addressed by our algorithm consists in making the following decisions:

- State of the i-th scalar input (inactive, active);
- State of the j-th functional input (inactive, active);
- Projection basis for the j-th functional input $(B_1, \ldots, B_z)$;
- Projection dimension for the j-th functional input $(0, \ldots, k_j)$;
- Distance for the j-th functional input $(D_1, \ldots, D_w)$;
- Kernel type $(K_1, \ldots, K_x)$,

with $i \in \{1, \dots ds\}$, $j \in \{1, \dots df\}$ and k_j the original dimension of input j. The sets $\{B_1, \dots, B_z\}$, $\{D_1, \dots, D_w\}$ and $\{K_1, \dots, K_x\}$ correspond to the basis, distance and kernel families to be considered, in that order. The projection dimension 0 denotes no projection. In order to find a suitable combination of the parameters listed above, we let our artificial ants to move through a network with a structure similar to the one depicted in Figure 1. Such a structure prevents the constitution of senseless solutions (e.g., an input being both, inactive and active) and helps to keep the network data structures considerably simple by only defining strictly necessary links.

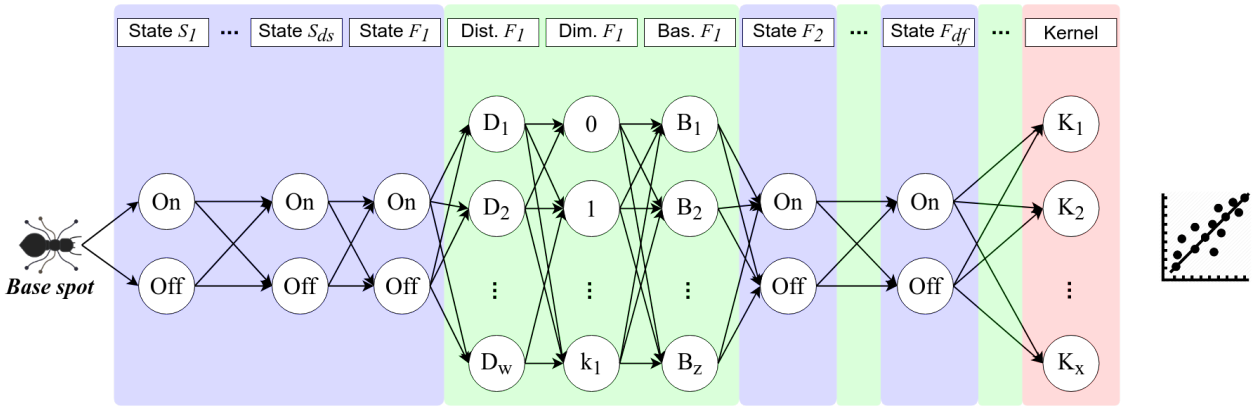


Figure 1: Decision network used by the ant colony based heuristic for model selection. One end-to-end path over the network provides a feasible set of structural parameters.

The implementation of the algorithm in `funGp` strictly considers the levels of kernel function, projection basis and distance type, listed in Sections 2.1, 2.2 and 2.4, respectively. However, both the foundations of the approach and the code implementations are general enough to be easily extended to other levels in future versions of the package. This manual does not go further into the methodological details of the algorithm, however, a detailed explanation of it is offered in [14] for the interested reader.

3.2 Using the model factory in `funGp`

In this section we explain how to manipulate the `fgpm_factory` function in order to get optimized model structures. The examples in this section are based on the analytic black-box function

$$
\begin{aligned}
\mathcal{G}_2: \quad & [0,1]^5 \times \mathcal{F}^2 \to \mathbb{R}, \\
& (\boldsymbol{x}, \boldsymbol{f}) \mapsto \left(x^{(2)} + 4x^{(3)} - \frac{5}{4\pi^2}\left(x^{(1)}\right)^2 + \frac{5}{\pi}x^{(1)} - 6 \right)^2 \\
& \quad + 10\left(1 - \frac{1}{8\pi}\right)\cos\left(x^{(1)}\right)\left(x^{(2)}\right)^2\left(x^{(5)}\right)^3 + 10 \\
& \quad + \frac{4}{3}\pi\left(42\sin\left(x^{(4)}\right)\int_0^1 15\ f^{(1)}(t)\,(1-t) - 5\ dt \right. \\
& \qquad \left. + \pi\left(\frac{x^{(1)}x^{(5)} + 5}{5} + 15\right)\int_0^1 15\ t\ f^{(2)}(t)dt\right),
\end{aligned}
$$

with $\boldsymbol{x} = \left(x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}, x^{(5)}\right)$ the scalar inputs, $\boldsymbol{f} = \left(f^{(1)}, f^{(2)}\right)$ the functional inputs,

and $\mathcal{F}$ the set of continuous functions from [0, 1] to $\mathbb{R}$. This function is inspired by the second analytic example studied in [3], with three additional scalar inputs allocated over the different terms of the equations to increase a bit its complexity. This function 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.

• Getting started

Let us open this section with a basic call to the factory using its default attribute values.

```
# generating input and output data
set.seed(100)
n.tr <- 32
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)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB7(sIn, fIn, n.tr)

# calling the funGp factory
xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut) # (~10 seconds)
```

The output of `fgpm_factory` is an object of class `Xfgpm`. It includes a variety of information on it that we will be explored later in detail, towards the end of this section. For now, let us concentrate on the **`@model`** slot, which contains the selected regression model. This is an object of type `fgpm`, which can be plotted using the `plotLOO` function. Just to illustrate, let us compare our optimized model with that obtained if we arbitrarily use the default argument values in the `fgpm` function, i.e., using `fgpm(sIn = sIn, fIn = fIn, sOut = sOut)`.

```
# plotting the optimized model
plotLOO(xm@model)

# plotting the model of default fgpm structural configuration
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut)
plotLOO(m1)
```

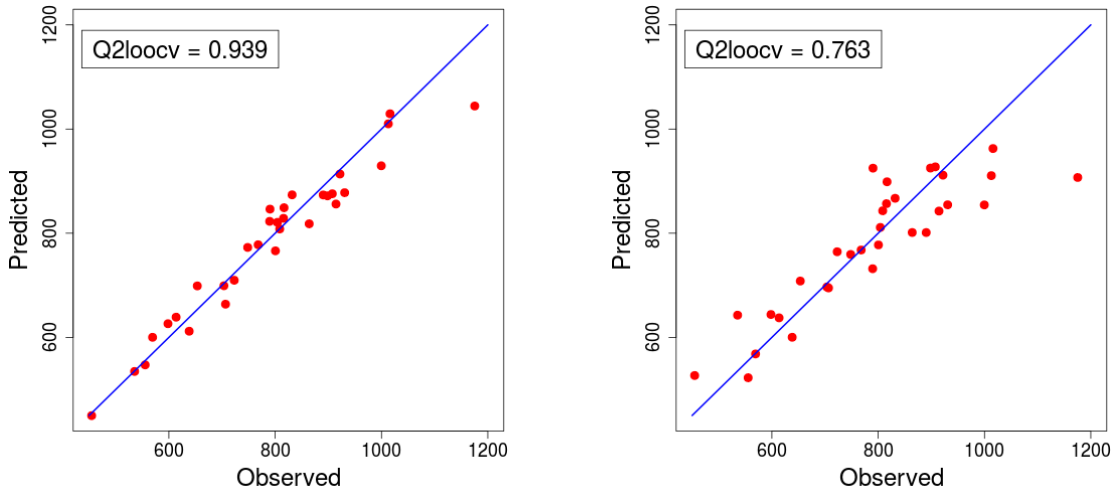


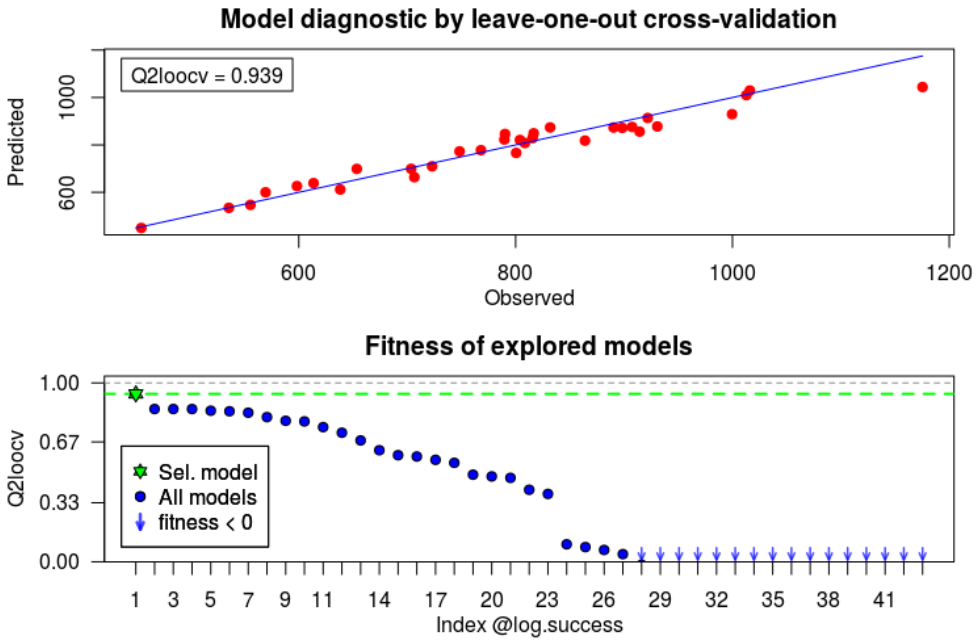
Figure 2: Calibration plot of two structural configurations for the same input and output data. Left panel: optimized configuration. Right panel: unoptimized, arbitrary configuration.

Right away, just by calling `fgpm_factory` with its default arguments, we were able to find a model of greater quality. Some key points in the light of this first result are:

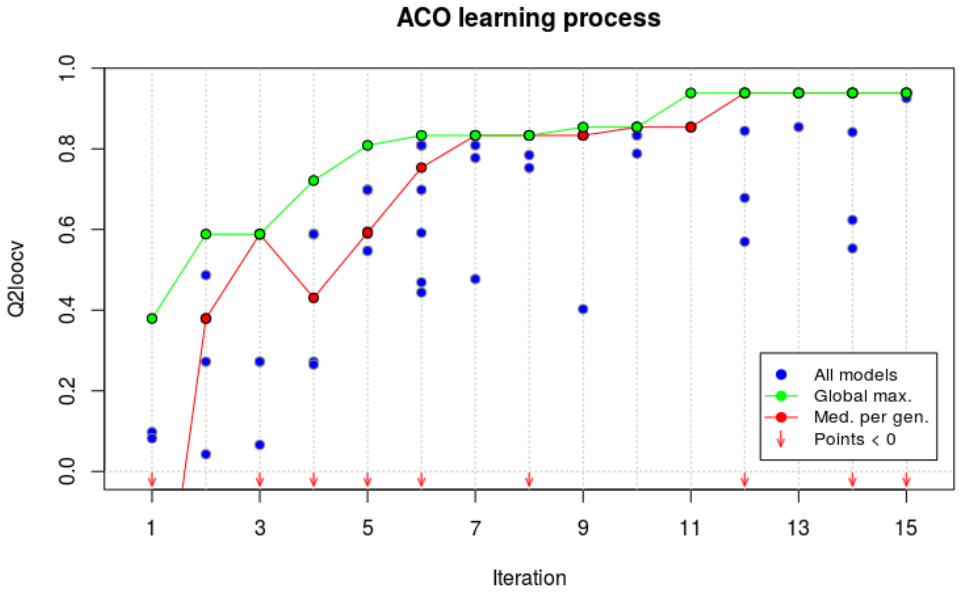
- Firstly, the superiority of the optimized model does not imply that the default argument values of the `fgpm` function are bad. They are just not tailored to this specific regression instance, contrarily to the structural parameters selected by `fgpm_factory`. That is the purpose of having `fgpm_factory` in the package, to be able to find good structural parameters for any regression instance that `fgpm` could handle.
- Secondly, the result does not mean that `funGp` models should always be made through `fgpm_factory`. In this example we see how the unoptimized model still presents a high Q^2_{loocv}. However, if there is time, we strongly recommend to perform the optimization.
- Finally, the superiority of the model delivered by `fgpm_factory` is exclusively fostered by the optimization of the structural parameter configuration, and has nothing to do with 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 explanation of the usage of `fgpm_factory`. The outputs of this function can be plotted by either the `plotX` or the `plotEvol` function. The former one provides a notion of the absolute and relative quality of the selected model, and the second one illustrates the evolution of the quality of the explored models along the iterations.

```
# displaying plots on the quality of the selected model
plotX(xm)
```



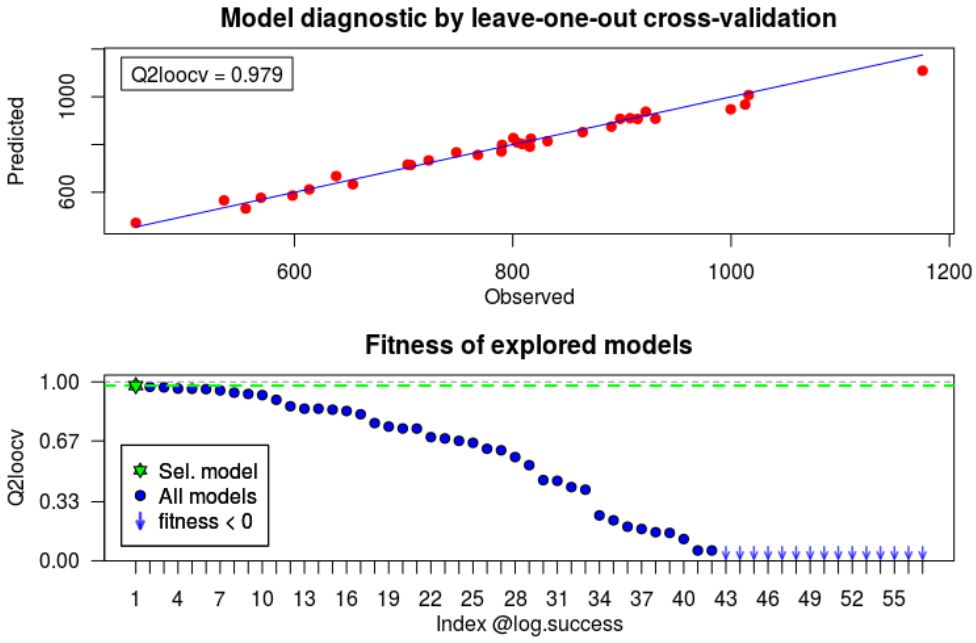

```
# plotting the evolution of the objective function
plotEvol(xm)
```



Even after multiple iterations, some points still fall relatively far from the maximum. This happens mainly because we have multiple categorical features, whose alteration might change the performance statistic in a nonsmooth way. Nonetheless, the median stays close to the maximum, which confirms that the exploration is converging towards the best known solutions. On the other hand, the points that fall bellow zero usually correspond to models whose hyperparameters were hard to optimize. This occurs sporadically during the log-likelihood optimization for Gaussian processes, due to the non-linearity of the objective function.

An easy way to improve the quality of the selected model is just to let the algorithm complete more iterations. This can be done through the argument **`setup`**, as below.

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



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

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

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

With this call, the 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} [8] is computed. This procedure ensures fairness in the comparison, since all the models use the same training and validation sets. In order to account for the sampling noise, the user may want to use multiple training-validation pairs of sets. This option is easily requested to the factory by passing a **`matrix`** instead of an **`array`** through the argument **`ind.vl`**. Such a **`matrix`** should have the indices for one training set on each column. This means, that the **`matrix`** should have as many rows as validation points, and as many columns as replicates.

```
# generating validation indices
ind.vl <- replicate(30, sample(seq_len(n.tr), 5)) # about 15% of points for validation, 30 replicates

# calling the funGp factory
xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, ind.vl = ind.vl) # (~4 minutes)
```

The larger the number of replicates, the longest the optimization will be, but also the less noise will appear on the statistics used to compare the models. Section 4 addresses the reduction of processing time through parallelization.

Note that the calibration plot produced by `plotX` will always report the Q^2_{loocv} statistic, regardless of whether this or the Q^2_{hout} was used for the optimization of the structural parameters. In contrast, the bottom frame will always display the statistic used during the optimization. When validation indices are provided, the model stored in the **`@model`** slot of the `Xfgpm` object will be one trained with as many points as remain once the specified validation points are removed. When multiple validation sets are specified, the model stored in the **`@model`** slot of the `Xfgpm` object will be selected in two steps by: (i) identifying the structural configuration of higher average Q^2_{hout}; and (ii) pick the replicate of best structural configuration with higher Q^2_{hout}.

• Setting up the parameters of the heuristic

Our model selection algorithm relies on a set of parameters typical of any ant colony based method. Roughly speaking, those parameters control the number of individuals and iterations, the degree of exploration and rate of convergence, along with the learning-reinforcement mechanism in the algorithm. The default values of those parameters in `funGp` were selected based on the values used by Dorigo et al. in the introductory paper of the Ant Colony System [15]. We validated the suitability of that setting for our model selection problem through a large set of trials involving different black-box functions like the one defined at the beginning

of this manual (Section 1), and more than 10 others that raised in the frame of the RISCOPE research project [16] (see [14] for more details). Here we explain how to modify the parameters of the heuristic in case the user wants to experiment with them. Our algorithm performs based on the following list of parameters:

Initial pheromone load

- **`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. **`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 (see Section 2.4) with the projection dimension*. **`Default: 2`** and **`1.4`**, respectively.

Local pheromone update

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

Global pheromone update

- **`u.gbest`**: the algorithm works in an iterative fashion; should the pheromone load on the links of the best ant so far over all the iterations be reinforced? **`Default: FALSE`**.
- **`n.ibest`**: the algorithm always reinforces the links of the best **`n.ibest`** ants of each iteration; how many ants should be considered for reinforcement? **`Default: 1`**.
- **`rho.g`**: learning reinforcement rate*. **`Default: 0.1`**.

Population factors

- **`n.iter`**: number of iterations. Each iteration involves the exploration of the solution space, constitution of a set of model configurations, evaluation of their performance in prediction and system feedback. **`Default: 15`**.
- **`n.pop`**: number of ants per iteration; each ant corresponds to one solution to the problem, in this case, a structural configuration for the model. **`Default: 10`**.

Bias strength

- **`q0`**: ants use one of two rules to select their next node at each step. The first rule leads the ant through the link with higher 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. They 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 facilitate the testing of each input in at least a few models.

The parameters marked with an asterisk (*) are explained more thoroughly in [14]. All the parameters listed above can be accessed in a `fgpm_factory` call through the argument **`setup`**, which should be a **`list`**. Below an example using arbitrary setup values.

```
# calling the funGp factory with an arbitrary setup
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)
xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, setup = mysup) # (~18 seconds)
```

• 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 an example.

```
# generating input and output data
set.seed(100)
n.tr <- 32
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)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB7(sIn, fIn, n.tr)

# setting up the constraints
myctr <- list(s_keepOn = c(1,2), # keep both scalar inputs always on
              f_keepOn = c(2), # keep f2 always active
              f_disTypes = list("2" = c("L2_byindex")), # only use L2_byindex distance for f2
              f_fixDims = matrix(c(2,4), ncol = 1), # f2 should be projected onto a space of dimension 4
              f_maxDims = matrix(c(1,5), ncol = 1), # f1 should be projected onto a space of dimension max 5
              f_basTypes = list("1" = c("B-splines")), # only use B-splines projection for f1
              kerTypes = c("matern5_2", "gauss")) # test only Matern 5/2 and Gaussian kernels

# calling the funGp factory with specific constraints
xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, ctraints = myctr) # (~15 seconds)
```

This call to the factory will exclusively explore models that fulfill the constraints passed through **`ctraints`**. This can be verified by inspecting the **`@log.success@sols`** slot of the `Xfgpm` object returned by `fgpm_factory`.

```
# checking log of some successfully built models
cbind(xm@log.success@sols, "Q2" = xm@log.success@fitness)
```

R output:

	State_X1	State_X2	State_X3	State_X4	State_X5	State_F1	Distance_F1	Dim_F1	Prj_basis_F1	State_F2	Distance_F2	Dim_F2	Prj_basis_F2	Kernel	Q2
1	On	On	Off	On	Off	Off	--	-	--	On	L2_byindex	4	B-splines	matern5_2	0.77
2	On	On	Off	On	Off	On	L2_byindex	3	B-splines	On	L2_byindex	4	B-splines	matern5_2	0.74
3	On	On	Off	On	On	Off	--	-	--	On	L2_byindex	4	B-splines	matern5_2	0.64
4	On	On	Off	On	On	On	L2_byindex	3	B-splines	On	L2_byindex	4	B-splines	matern5_2	0.47
5	On	On	On	On	Off	Off	--	-	--	On	L2_byindex	4	B-splines	matern5_2	0.43
6	On	On	Off	Off	On	Off	--	-	--	On	L2_byindex	4	B-splines	gauss	0.43
7	On	On	Off	Off	Off	Off	--	-	--	On	L2_byindex	4	B-splines	matern5_2	0.42
8	On	On	On	On	On	On	L2_byindex	1	B-splines	On	L2_byindex	4	B-splines	matern5_2	0.38
9	On	On	On	On	On	Off	--	-	--	On	L2_byindex	4	B-splines	matern5_2	0.27
10	On	On	On	On	On	Off	--	-	--	On	L2_byindex	4	PCA	gauss	0.26
11	On	On	On	On	On	Off	--	-	--	On	L2_byindex	4	PCA	matern5_2	0.12
12	On	On	On	Off	On	Off	--	-	--	On	L2_byindex	4	B-splines	matern5_2	0.10
13	On	On	Off	Off	On	On	L2_byindex	1	B-splines	On	L2_byindex	4	B-splines	gauss	0.02
14	On	On	Off	Off	On	On	L2_byindex	2	B-splines	On	L2_byindex	4	B-splines	matern5_2	-0.05
15	On	On	Off	On	Off	Off	--	-	--	On	L2_byindex	4	PCA	matern5_2	-0.07
16	On	On	Off	On	Off	Off	--	-	--	On	L2_byindex	4	PCA	gauss	-0.10
17	On	On	Off	Off	On	On	L2_byindex	3	B-splines	On	L2_byindex	4	B-splines	matern5_2	-0.16
18	On	On	On	Off	On	On	L2_bygroup	3	B-splines	On	L2_byindex	4	PCA	gauss	-0.27

• Time based stopping condition

The basic stopping condition for any ant colony based algorithm is the number of iterations. This type of stopping condition is often useful during the development stage of the algorithm.

However, in the wild it is hard to know in advance which number of iterations will be suitable for the problem at hand, and even if one had an idea, it would still be difficult to estimate how much processing time that would suppose. In practice we recommend to use instead a time based stopping condition. It works by defining a time budget for structural optimization, and then letting the heuristic run until the budget is exhausted. This possibility has been implemented in `fgpm_factory`, and is accessible through the **`time.lim`** argument.

```
# setting up a sufficiently large number of iterations
mysup <- list(n.iter = 2000)

# defining time limit
mytlim <- 60

# calling the funGp factory with time based stopping condition
xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, setup = mysup, time.lim = mytlim)

R output:
** Time limit reached, exploration stopped after 60.01 seconds. # 163 iterations done
```

When using the time based stopping condition, the number of iterations should be set sufficiently large so that it does not cause a premature stop of the exploration. The argument **`time.lim`** should always be provided in seconds. Once the time limit is reached, the algorithm will attempt to stop as soon as possible, however, the ongoing training process of a model will never be interrupted. Thus, the actual processing time will normally exceed the specified time budget for a bit. This discrepancy might be more noticeable for problems involving heavier model configurations with larger number of inputs or a larger amount of data.

• Further exploring the `Xfgpm` object

After checking different things that can be done through a `fgpm_factory` call, it is good time to dedicate some attention to the information contained in the object delivered by the function. The object is of class `Xfgpm`, which includes diverse information about the selected model and also about the model selection process carried on. Below a list of the slots of the object with a short description of each.

Selected model

- **`@model`**: selected model delivered by the `fgpm` function.
- **`@structure`**: **`data.frame`** with the selected structural configuration.
- **`@stat`** and **`@fitness`**: type and value of the performance statistic used for the optimization of the structural parameters. Currently, the type of performance statistic can be either Q^2_{loocv} or Q^2_{hout} (see Sections 1.1 and 1.2 for details on these measures).

Record of explored models

- **`@log.success`**: object of class `antsLog` with the structure, function calls and performance statistic of all models successfully made during the optimization, organized in decreasing order of performance.
- **`@log.crashes`**: object of class `antsLog` with the structure and function calls of all models whose `fgpm` function call crashed.

Exploration extent

- **@n.solspace**: total number of structural configurations that could be made, based on the specified solution space.
- **@n.explored**: total number of structural configurations successfully built and evaluated during the exploration.

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.

By conducting the structural optimization through fgpm_factory, one obtains not only one but a set of high quality models. Those are accessible through the **@log.success@sols** and **@log.success@args** slots. The former contains a data frame with all the levels of structural parameters selected for each explored model. This data structure could be useful to make a posterior analysis on patterns that lead to high quality models. The **@log.success@args** slot contains exactly the same information, but in a format that allows the easy reconstruction of any of the explored models. We illustrate this possibility with the following example. We start by performing a structural optimization.

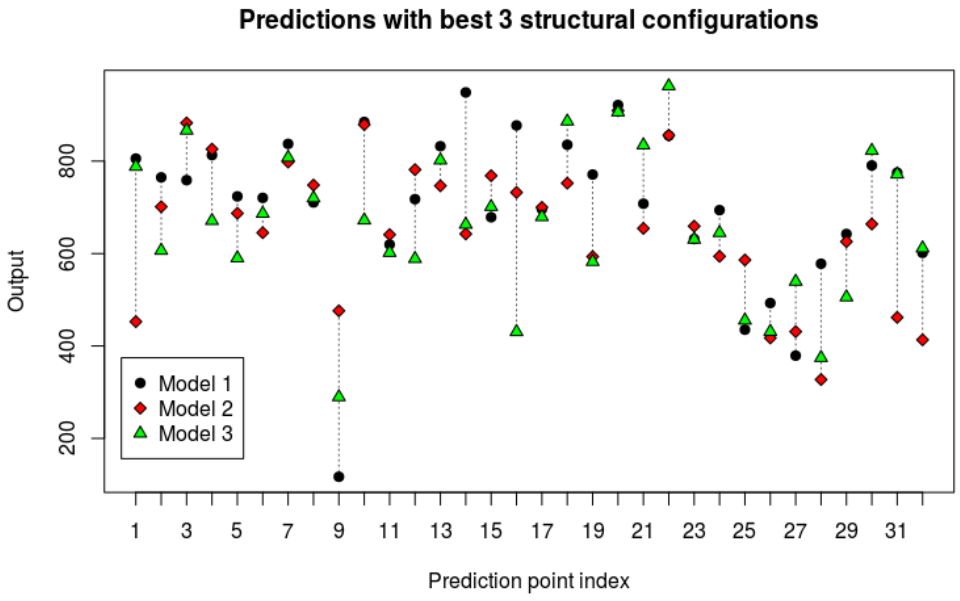
```
# generating input and output data
set.seed(100)
n.tr <- 32
sIn <- expand.grid(x1 = seq(0,1,length = n.tr^(1/5)), x2 = seq(0,1,length = n.tr^(1/5)),
                   x3 = seq(0,1,length = n.tr^(1/5)), x4 = seq(0,1,length = n.tr^(1/5)),
                   x5 = seq(0,1,length = n.tr^(1/5)))
fIn 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

# calling the funGp factory
xm <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut) # (~10 seconds)
```

After some time we update our dataset. Now we have 243 points instead of 32. Then, we rebuild the best three models using the new data.

```
# generating new data
n.tr <- 243 # more points!
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)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB7(sIn, fIn, n.tr)

# re-building the three best models based on the new data (one model at a time)
# m1 <- eval(parse(text = xm@log.success@args[[1]]@string)[[1]])
# m2 <- eval(parse(text = xm@log.success@args[[2]]@string)[[1]])
# m3 <- eval(parse(text = xm@log.success@args[[3]]@string)[[1]])

# re-building the three best models based on the new data (compact code with all 3 calls)
modStack <- lapply(1:3, function(i) eval(parse(text = xm@log.success@args[[i]]@string)[[1]]))
```

Finally, we use each model for prediction. Here, the format4pred function will generate a list with the scalar and functional inputs to use for each model. If any of the two types of inputs is not present in the model, format4pred will set it to **NULL**, which will be properly interpreted by the fgpm function.

```
# extracting the fgpm arguments of the three best models
argStack <- xm@log.success@args[1:3]

# generating input data for prediction
n.pr <- 32
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)))
  fIn.pr <- list(f1 = matrix(runif(n.pr*10), ncol = 10), matrix(runif(n.pr*22), ncol = 22))

# making predictions based on the three best models (compact code with all 3 calls)
preds <- do.call(cbind, Map(function(model, args) {
  in4matted <- format4pred(sIn.pr = sIn.pr, fIn.pr = fIn.pr, args)
  predict(model, sIn.pr = in4matted$sIn.pr, fIn.pr = in4matted$fIn.pr)$mean
}, modStack, argStack))

# plotting predictions made by the three models
require(plyr) # for conciseness
plot(1, xlim = c(1,nrow(preds)), ylim = range(preds), xaxt = "n",
     xlab = "Prediction point index", ylab = "Output",
     main = "Predictions with best 3 structural configurations")
axis(1, 1:nrow(preds))
l_ply(seq_len(n.pr), function(i) lines(rep(i,2), range(preds[i,1:3]), col = "grey35", lty = 3))
points(preds[,1], pch = 21, bg = "black")
points(preds[,2], pch = 23, bg = "red")
points(preds[,3], pch = 24, bg = "green")
legend("bottomleft", legend = c("Model 1", "Model 2", "Model 3"),
       pch = c(21, 23, 24), pt.bg = c("black", "red", "green"), inset = c(.02,.08))
```



4 Parallelization in `funGp`

Sections 1, 2 and 3 made a good description of `funGp` from the perspective of functionality. This last section focuses on efficiency. Both, the `fgpm` and `fgpm_factory` functions have been equipped with the ability to exploit the existence of parallel environments. Below we explain how to use this feature.

4.1 Parallelized hyperparameters optimization

Let us start with the fgpm function, used to create regression models (see Section 1.1). In **funGp**, the selection of the hyperparameters of the model is made by likelihood maximization. For Gaussian processes, 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 in local optima. This can be requested to fgpm through the argument **n.starts**, which should be assigned an integer value corresponding to the number of starting points to use. Below an example using 10 starting points.

```
# generating input data for training
set.seed(100)
n.tr <- 243
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)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))

# generating output data for training
sOut <- fgp_BB7(sIn, fIn, n.tr)

# calling fgpm with multistart in sequence
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, n.starts = 10) # (~22 seconds)
```

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.

```
# calling fgpm with multistart in parallel
cl <- parallel::makeCluster(3)
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, n.starts = 10, par.clust = cl) # (~14 seconds)
parallel::stopCluster(cl)
```

As a good practice, the cluster must be stopped right after finishing the requested task in order to prevent memory issues.

Remark: in order to provide progress bars for the monitoring of time consuming processes ran in parallel, **funGp** relies on the **doFuture** [17] and **future** [18] R packages. Unfortunately, under this setting, parallel processes suddenly interrupted tend to leave corrupt connections that will show up as an error next time you try to perform the parallelized task. To make it clear, if you launch fgpm in parallel and you stop the process by hand, before it ends, and then you try to repeat the call in parallel, you may likely find and error indicating that **... the connection to the worker is corrupt...**. If that happens to you, the following workaround will help to regain control of parallel processing. Once you get the error, repeat the function call using a different number of nodes. For instance, let us assume that you had run with 3 nodes in the call that produced the error. We can make the new function call, for instance with 2 nodes.

```
# repeating the call with different number of nodes
cl <- parallel::makeCluster(2)
m1 <- fgpm(sIn = sIn, fIn = fIn, sOut = sOut, n.starts = 10, par.clust = cl)
parallel::stopCluster(cl)
```

There is no need to let this process become complete, you can stop it by hand a couple seconds after making the function call. That is it. Now you can launch again the process

in parallel with the number of nodes that you were originally using. We acknowledge that this is more a trick than an ideal way to resolve this types of issues. However, this problem is originated outside **funGp**, which limits our control over it. We find the approach shown above a pragmatic solution for most users. We will remain attentive in case it appears a more elegant solution to this problem. All this discussion also applies for parallelized calls to `fgpm_factory`, which will be discussed in the next section.

4.2 Parallelized model selection

Parallelization is also present in the model factory. Each ant in our heuristic algorithm represents a structural configuration, and eventually translates into a regression model. Each ant influence on the decisions made by the others since they share a common decision network and all of them affect the pheromone load in the links. Nonetheless, each time all the ants of one iteration complete a model structure, each of the models is built and evaluated for performance in an independent fashion. Thus, once all the structural configurations of one iteration are complete, the construction of the corresponding models is a task that can be performed in parallel. The way to do that is identical to how it is done in the `fgpm` function. For this, the user must define a parallel processing cluster and then pass it to `fgpm_factory` through the **`par.clust`** argument, as below.

```
# generating input and output data
set.seed(100)
n.tr <- 243
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)))
fIn <- list(f1 = matrix(runif(n.tr*10), ncol = 10), f2 = matrix(runif(n.tr*22), ncol = 22))
sOut <- fgp_BB7(sIn, fIn, n.tr)

# calling fgpm_factory in parallel
cl <- parallel::makeCluster(3)
xm.par <- fgpm_factory(sIn = sIn, fIn = fIn, sOut = sOut, par.clust = cl) # (~200 seconds)
parallel::stopCluster(cl)
```

The advice given when explaining the parallelization in the `fgpm` function applies here; the cluster must be stopped right after finishing the requested task in order to prevent memory issues. In addition, we clarify that parallelized processing should be reserved for cases where each individual process (call to `fgpm`) takes a significant amount of time. If the call in sequence is already long due to the large number of processes it involves, but each process runs almost immediately, the benefit of parallelization might become null. Thus, we prescribe the use of this feature for problems where the evaluation of a single model takes several seconds or more. In such a context, parallelization will allow the evaluation of a larger number of structural configurations in the same amount of time.

Closing discussion

funGp started as a set of scripts enabling to include functional inputs in a regression model. What we present in this tutorial is that and much more. We have done our best to provide a powerful regression tool for all-level users. No impositions on the type of relationship between inputs and outputs, no need of pre-processing of the functional inputs, no need for complex data structures. You put the data and we put the power of the Gaussian process models in order to efficiently extract the underlying information from it. The more expertise

the user has in statistics and also in the usage of the package, the more it will be able to discover new possibilities and features. We make strong emphasis on the model selection functionality, which takes models' construction to a whole new level. Any regression package gives you a model in response for your data. Some packages return different types of models depending on your specifications. Not very often a package helps you to chose the good model, and this is what `funGp` does. During the implementation, we kept present at all time the need for efficiency, and we made an effort to make everything run fast and smooth. Parallelization in the `fgpm` and `fgpm_factory` functions is a valuable commodity in this regard. We envisage the extension of the package in multiple different aspects, and therefore, we made the implementations with scalability in mind. All the structural parameters are modifiable in order to include additional levels or even other structural parameters than those currently available. The heuristic model selection algorithm was also designed to be easily adaptable to this type of extension. Going further, the `fgpm_factory` function was structured in such a way that other model selection methods could be added later. Being `funGp` a piece of open source, we encourage the community to make contributions in any line found pertinent.

Acknowledgements

This study was conducted in the frame of the RISCOPE project, funded by the French Agence Nationale de la Recherche (ANR). We thank the ANR for this support. We are also grateful to Yves Deville from Alpestat for his advice on documentation of R packages and to Juliette Garcia from ENAC for her assistance on the stabilization of the Ant Colony algorithm for structural parameter optimization.

References

[1] J. Betancourt, F. Bachoc, T. Klein, D. Idier, R. Pedreros, and J. Rohmer, "Gaussian process metamodeling of functional-input code for coastal flood hazard assessment," *Reliability Engineering & System Safety*, p. 106870, 2020.

[2] C. Lataniotis, S. Marelli, and B. Sudret, "Extending classical surrogate modelling to ultrahigh dimensional problems through supervised dimensionality reduction: a data-driven approach," *arXiv preprint arXiv:1812.06309*, 2018.

[3] T. Muehlenstaedt, J. Fruth, and O. Roustant, "Computer experiments with functional inputs and scalar outputs by a norm-based approach," *Statistics and Computing*, vol. 27, no. 4, pp. 1083–1097, 2017.

[4] B. D. Ripley, *Spatial statistics*, vol. 575. John Wiley & Sons, 2005.

[5] O. Dubrule, "Cross validation of kriging in a unique neighborhood," *Journal of the International Association for Mathematical Geology*, vol. 15, no. 6, pp. 687–699, 1983.

[6] J. O. Ramsay and B. W. Silverman, *Applied functional data analysis: methods and case studies*. Springer, 2007.

[7] A. Marrel, B. Iooss, M. Jullien, B. Laurent, and E. Volkova, "Global sensitivity analysis for models with spatially dependent outputs," *Environmetrics*, vol. 22, no. 3, pp. 383–397, 2011.

[8] J. Nilsson, S. de Jong, and A. K. Smilde, "Multiway calibration in 3d qsar," *Journal of Chemometrics: A Journal of the Chemometrics Society*, vol. 11, no. 6, pp. 511–524, 1997.

[9] C. De Boor, "A practical guide to splines," in *Applied mathematical sciences*, vol. 27, pp. 15–16, Heidelberg: Springer, 1978.

[10] I. Jolliffe, *Principal component analysis*. Springer, 2011.

[11] I. Papaioannou, M. Ehre, and D. Straub, "Pls-based adaptation for efficient pce representation in high dimensions," *Journal of Computational Physics*, vol. 387, pp. 186–204, 2019.

[12] A. Cohen, I. Daubechies, and J.-C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications on pure and applied mathematics*, vol. 45, no. 5, pp. 485–560, 1992.

[13] B. Schölkopf, A. Smola, and K.-R. Müller, "Nonlinear component analysis as a kernel eigenvalue problem," *Neural computation*, vol. 10, no. 5, pp. 1299–1319, 1998.

[14] J. Betancourt, F. Bachoc, T. Klein, and Gamboa, "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, Apr. 2020. hal-02532713.

[15] M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on evolutionary computation*, vol. 1, no. 1, pp. 53–66, 1997.

[16] "ANR RISCOPE Project." https://perso.math.univ-toulouse.fr/riscope/. Accessed: 2020-03-15.

[17] H. Bengtsson, *doFuture: A Universal Foreach Parallel Adapter using the Future API of the 'future' Package*, 2020. R package version 0.9.0.

[18] H. Bengtsson, *future: Unified Parallel and Distributed Processing in R for Everyone*, 2020. R package version 1.16.0.