funMoDisco: Functional Motif Discovery

Niccolò Feresini

2024-11-01

Contents

Introduction

The `discoverMotifs` function serves as the core of the package, offering a robust and efficient implementation of two advanced algorithms: **ProbKMA**(Cremona and Chiaromonte, 2020) and **FunBiAlign**(Di Iorio, Cremona,and Chiaromonte, 2023). Together, these algorithms provide a comprehensive solution for the detection and clustering of recurring patterns within functional data. In addition to motif discovery, `funMoDisco` allows users to **simulate functional curves with embedded motifs**. The package provides several functions for generating synthetic functional data, which can be useful for testing and benchmarking the motif discovery algorithms. These simulated curves are customizable, allowing users to control the number, length, and complexity of the motifs. By the end of this vignette, users will be guide through practical examples and equipped with a solid understanding of how to effectively utilize `funMoDisco` package for their pattern detection needs.

Overview of discoverMotifs

The `discoverMotifs` function allows users to:

- Choose to run **ProbKMA** multiple times with varying numbers of motifs (`K`) and minimum motif lengths (`c`), or **FunBiAlign** specifying the length (`portion_len`) and the minimum cardinality (`min_card`) of the motifs.
- Perform clustering based on local alignments of curve segments.
- Control the clustering process through a wide range of hyperparameters.

Common Parameters

The following parameters are shared between both algorithms:

Parameter	Description	Default
Y0	A list of N vectors (for univariate curves) or N matrices (for multivariate curves) representing the curves, where each curve is evaluated on a uniform grid.	mandatory
method	A character string specifying which method to use: either 'ProbKMA' or 'funBIalign'.	mandatory
stopCriterion	A character string indicating the convergence criterion based on Bhattacharyya distance between memberships for 'ProbKMA', or ranking criteria for 'funBIalign'.	mandatory
name	A character string providing the name of the resulting folder.	mandatory
plot	A logical value indicating whether to plot the motifs and results.	mandatory
worker_number	An integer specifying the number of CPU cores to use for parallel computations. Defaults to cores minus one.	'detectCores() - 1'

Key Arguments for ProbKMA

Below is an overview of the key arguments for the `ProbKMA` algorithm in the `discoverMotifs` function:

Parameter	Description	Default
K	A vector specifying the numbers of motifs to be tested.	mandatory
c	A vector specifying the minimum motif lengths to be tested.	mandatory
diss	Dissimilarity. Possible choices are 'd0_L2', 'd1_L2', 'd0_d1_L2'.	mandatory
alpha	Parameter in [0,1] defining the relative weight of the curve's levels and derivatives. 'alpha'=0 means 'd0_L2', 'alpha'=1 means 'd1_L2'.	mandatory
Y1	A list of N vectors or matrices representing the derivative of the curves. Required if 'diss='d0_d1_L2''.	'NULL'
P0	A matrix specifying the initial membership probabilities. If not specified, it will be randomly generated.	'matrix()'
S0	A matrix specifying the initial shift. If not specified, it will be randomly generated.	'matrix()'

c_max	An integer or a vector of K integers specifying the maximum motif lengths.	‘Inf‘
w	Weight vector for the dissimilarity index across dimensions.	‘1‘
m	Weighting exponent in the least-squares functional method (must be greater than 1).	‘2‘
iter_max	Maximum number of iterations allowed for the ProbKMA algorithm.	‘1e3‘
quantile	Double specifying quantile probability when ’stopCriterion’="quantile".	‘0.25‘
tol	Double specifying the tolerance level for the method; iteration stops if the stop criterion is less than ’tol’.	‘1e-8‘
iter4elong	Integer specifying the number of iterations after which motif elongation is performed. If ’iter4elong’ > ’iter_max’, no elongation is performed.	‘100‘
tol4elong	Tolerance on the Bhattacharyya distance for motif elongation.	‘1e-3‘
max_elong	Maximum elongation allowed in a single iteration, as a percentage of motif length.	‘0.5‘
trials_elong	Integer specifying the number of elongation trials (equispaced) on each side of the motif in a single iteration.	‘201‘
deltaJK_elong	Maximum relative increase in the objective function allowed during motif elongation.	‘0.05‘
max_gap	Double specifying the maximum gap allowed in each alignment as a percentage of the motif length.	‘0.2‘
iter4clean	Integer specifying number of iterations after which motif cleaning is performed. If ’iter4clean’ > ’iter_max’, no cleaning is performed.	‘50‘
tol4clean	Tolerance on the Bhattacharyya distance for motif cleaning.	‘1e-4‘
quantile4clean	Dissimilarity quantile used for motif cleaning.	‘0.5‘
return_options	If ’TRUE’, the options passed to the method are returned.	‘TRUE‘
n_subcurves	Integer specifying the number of splitting subcurves used when the number of curves is equal to one.	‘10‘
sil_threshold	Double specifying the threshold value to filter candidate motifs.	‘0.9‘
set_seed	If ’TRUE’, sets a random seed to ensure reproducibility.	‘FALSE‘

seed	The random seed for initialization (used if set_seed=TRUE).	'1'
exe_print	If 'TRUE' and worker_number is equal to one, prints execution details for each iteration.	'FALSE'
transformed	A logical value indicating whether to normalize the curve segments to the interval [0,1] before applying the dissimilarity measure.	'NULL'
V_init	A list of motif sets provided as specific initializations for clustering rather than using random initializations.	'NULL'
n_init_motif	The number of initial motif sets from 'V_init' to be used directly as starting points in clustering.	'NULL'

Example Usage

Here is an example showing a possible use of `discoverMotifs` with **ProbKMA** :

```
library(funMoDisco)

diss <- 'd0_d1_L2'
alpha <- 0.5
# run probKMA multiple times (2x3x10=60 times)
K <- c(2,3)
c <- c(61,51)
n_init = 10

data("simulated200") # load simulated data

results = funMoDisco::discoverMotifs(
  Y0 = simulated200$Y0,
  method = "ProbKMA",
  stopCriterion = "max",
  name = './results_ProbKMA_VectorData/',
  plot = TRUE,
  probKMA_options = list(
    Y1 = simulated200$Y1,
    K = K,
    c = c,
    n_init = n_init,
    diss = diss,
    alpha = alpha
  ),
```

```
  worker_number = NULL
)
```

In this scenario, all plots generated during the algorithm's execution will be saved in the folder specified by the 'name' parameter. Additionally, the function handles the entire post-processing phase, including filtering patterns and searching for occurrences within the curves, presenting both intermediate and final results along with their corresponding plots. If the user chooses to only perform the post-processing by adjusting parameters like 'sil_threshold,' it is sufficient to call the same discoverMotifs function with the updated parameters. The algorithm will automatically load the previously computed results (which are computationally expensive) and proceed with the post-processing, returning updated plots and results.

Below is an example of how to call `discoverMotifs` with a customized `V_init` and `transformed = TRUE`:

```
library(funMoDisco)

# Define parameters
c <- 5
K <- 2
n_init <- 2
diss <- 'd0_d1_L2'
alpha <- 0.5

# Load sample data
data("simulated200")

motif1 <- list(v0 = matrix(runif(c * 4), nrow = c, ncol = 1))
motif2 <- list(v0 = matrix(runif(c * 4), nrow = c, ncol = 1))

# Optional: Include `v1` if using a dissimilarity measure that requires it
motif1$v1 <- matrix(runif(c * 4), nrow = c, ncol = 1)
motif2$v1 <- matrix(runif(c * 4), nrow = c, ncol = 1)

# Define V_init with multiple initial motif sets, matching `K` motifs
# per initialization
V_init <- list(
  list(motif1, motif2),   # Initialization 1 with 2 motifs
  list(motif1, motif2)    # Initialization 2 with 2 motifs
)

# Run discoverMotifs
results <- funMoDisco::discoverMotifs(
  Y0 = simulated200$Y0,
  method = "ProbKMA",
  stopCriterion = "max",
  name = './results_ProbKMA_VectorData/',
  plot = TRUE,
```

```
  probKMA_options = list(
    Y1 = simulated200$Y1, K = K, c = c, n_init = n_init,
    diss = diss, alpha = alpha, sil_threshold = 0.5,
    V_init = V_init,
    transformed = TRUE
  ),
  worker_number = NULL
)
```

Key Arguments for funBIalign

Below is an overview of the key arguments for the `ProbKMA` algorithm in the `discoverMotifs` function:

Parameter	Description	Default
K	A vector specifying the numbers of motifs to be tested.	mandatory
c	A vector specifying the minimum motif lengths to be tested.	mandatory
diss	Dissimilarity. Possible choices are 'd0_L2', 'd1_L2', 'd0_d1_L2'.	mandatory
alpha	Parameter in [0,1] defining the relative weight of the curve's levels and derivatives. 'alpha'=0 means 'd0_L2', 'alpha'=1 means 'd1_L2'.	mandatory
Y1	A list of N vectors or matrices representing the derivative of the curves. Required if 'diss='d0_d1_L2''.	'NULL'
P0	A matrix specifying the initial membership probabilities. If not specified, it will be randomly generated.	'matrix()'
S0	A matrix specifying the initial shift. If not specified, it will be randomly generated.	'matrix()'
c_max	An integer or a vector of K integers specifying the maximum motif lengths.	'Inf'
w	Weight vector for the dissimilarity index across dimensions.	'1'
m	Weighting exponent in the least-squares functional method (must be greater than 1).	'2'
iter_max	Maximum number of iterations allowed for the ProbKMA algorithm.	'1e3'

quantile	Double specifying quantile probability when 'stopCriterion'="quantile".	'0.25'
tol	Double specifying the tolerance level for the method; iteration stops if the stop criterion is less than 'tol'.	'1e-8'
iter4elong	Integer specifying the number of iterations after which motif elongation is performed. If 'iter4elong' > 'iter_max', no elongation is performed.	'100'
tol4elong	Tolerance on the Bhattacharyya distance for motif elongation.	'1e-3'
max_elong	Maximum elongation allowed in a single iteration, as a percentage of motif length.	'0.5'
trials_elong	Integer specifying the number of elongation trials (equispaced) on each side of the motif in a single iteration.	'201'
deltaJK_elong	Maximum relative increase in the objective function allowed during motif elongation.	'0.05'
max_gap	Double specifying the maximum gap allowed in each alignment as a percentage of the motif length.	'0.2'
iter4clean	Integer specifying number of iterations after which motif cleaning is performed. If 'iter4clean' > 'iter_max', no cleaning is performed.	'50'
tol4clean	Tolerance on the Bhattacharyya distance for motif cleaning.	'1e-4'
quantile4clean	Dissimilarity quantile used for motif cleaning.	'0.5'
return_options	If 'TRUE', the options passed to the method are returned.	'TRUE'
n_subcurves	Integer specifying the number of splitting subcurves used when the number of curves is equal to one.	'10'
sil_threshold	Double specifying the threshold value to filter candidate motifs.	'0.9'
set_seed	If 'TRUE', sets a random seed to ensure reproducibility.	'FALSE'
seed	The random seed for initialization (used if set_seed=TRUE).	'1'
exe_print	If 'TRUE' and worker_number is equal to one, prints execution details for each iteration.	'FALSE'

transformed	A logical value indicating whether to normalize the curve segments to the interval [0,1] before applying the dissimilarity measure.	'NULL'
V_init	A list of motif sets provided as specific initializations for clustering rather than using random initializations.	'NULL'
n_init_motif	The number of initial motif sets from 'V_init' to be used directly as starting points in clustering.	'NULL'

Example Usage

Here is an example showing a possible use of `discoverMotifs` with **funBIalign** :

```
library(funMoDisco)

data("simulated200") # load simulated data

funBialignResult <- funMoDisco::discoverMotifs(
  Y0 = simulated200$Y0,
  method = "FunBIalign",
  stopCriterion = 'fMRS',
  name = './results_FunBialign',
  plot = TRUE,
  funBIalign_options = list(
    portion_len = 60,
    min_card = 3,
    cut_off = 1.0
  )
)
```

As previously discussed for 'ProbKMA', if the user intends to execute only the post-processing phase related to the re-ranking of discovered motifs, they can simply call the same function, specifying the updated re-ranking criterion and, if necessary, adjusting the new cut_off value.

Motif Simulation

As previously noted, the package offers the capability to generate synthetic curves embedded with patterns. This functionality facilitates the testing of both algorithms and provides a reliable reference benchmark for performance evaluation.

The algorithm begins by generating random curves utilizing B-splines as the foundational tools. Subsequently, it incorporates either random or positional patterns into these curves. Finally, noise is introduced, which can manifest as either pointwise noise or noise applied to the expansion coefficients of the B-splines. This process effectively simulates real-world scenarios in which each measurement is associated with a degree of noise.

Key Arguments for motifSimulationBuilder

'motifSimulationBuilder' represents the first function to be called. In particular, it represents the constructor of the S4 class 'motifSimulation'.

Below is an overview of the key arguments:

Parameter	Description	Default
N	The number of background curves to be generated.	mandatory
len	The length of the background curves.	mandatory
mot_details	A list outlining the definitions of the motifs to be included. Each motif is characterized by its length, a set of coefficients that may be optionally specified, and the number of occurrences. These occurrences can be indicated either by specific positions within the curves or by a total count. In the latter case, the algorithm will randomly position the motifs throughout the curves.	mandatory
norder	Integer specifying the order of the B-splines.	3
coeff_min	Additive coefficients to be incorporated into the generation of coefficients for the background curves.	'-15'
coeff_max	Additive coefficients to be incorporated into the generation of coefficients for the background curves.	'15'
dist_knots	Integer specifying the distance between two consecutive knots.	'10'
min_dist_motifs	Integer specifying the minimum distance between two consecutive motifs embedded in the same curve.	''norder' * 'dist_knots''
distribution	Distribution from which the coefficients of the background curves are generated. You can choose between a uniform distribution or a beta distribution. Alternatively, you can pass a vector representing the empirical distribution from which you wish to sample.	'unif'

Key Arguments for generateCurves

After calling the constructor of the class, it is then possible to generate the curves with the motifs embedded.

Below is an overview of the key arguments:

Parameter	Description	Default
object	The S4 object first constructed.	mandatory
noise_type	A string specifying whether to add pointwise error or coefficients ('pointwise' and 'coeff').	mandatory
noise_str	A list corresponding to the number of motifs, specifying the structure of noise to be added for each motif. If 'pointwise' is chosen, the user can specify a list of vectors or matrices indicating the amount of noise for each motif. If 'coeff' is selected, a list of individual values or vectors can be provided.	mandatory
seed_background	An integer specifying the seed for background curve generation.	'777'
seed_motif	An integer specifying the seed for motif generation.	'43213'
only_der	If 'FALSE', a vertical shift is added to each motif instance.	'TRUE'
coeff_min_shift	Minimum vertical shift.	'-10'
coeff_max_shift	Maximum vertical shift.	'10'

Key Arguments for plot_motifs

This is the final function to be called. As indicated by its name, it generates summary plots. Each plot displays the background curve, the motif without noise, and the motif with noise highlighted within a shaded region.

Below is an overview of the key arguments:

Parameter	Description	Default
object	The S4 object first constructed.	mandatory
curves	The result of the previous method.	mandatory
path	Path specifying the directory where the results will be saved.	mandatory

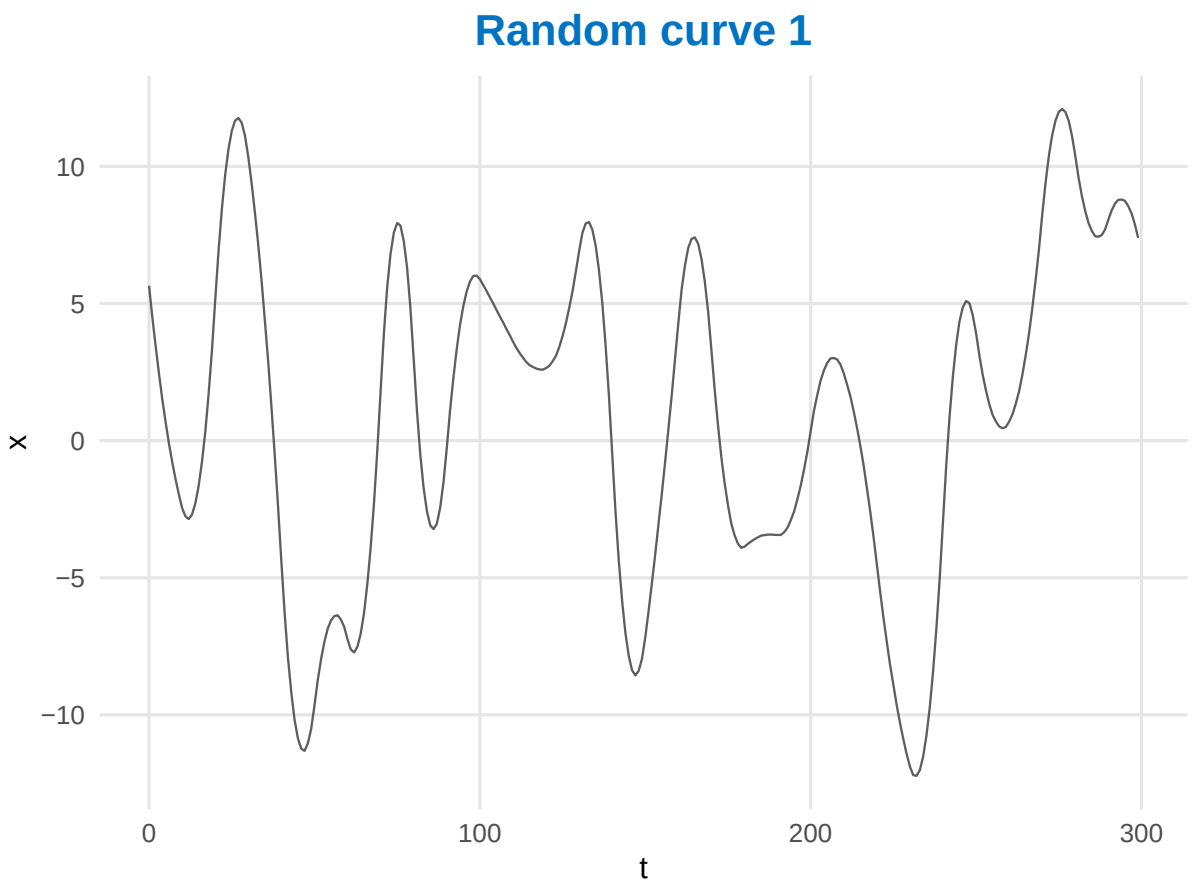
Examples

The five main types of use are considered below.

0) Special case: No motifs

```
library(funMoDisco)
```

```
mot_len <- 100
mot_details <- NULL # or list()

builder <- funMoDisco::motifSimulationBuilder(N = 20,len = 300,mot_details)

curves <- funMoDisco::generateCurves(builder)

funMoDisco::plot_motifs(builder,curves,name = "plots_0")
#> pdf
#>   2
```



1) Set the motif position and add pointwise noise

```
library(funMoDisco)

mot_len <- 100

# Struct specifying the motif ID, the number of curves, and the relative knot position
motif_str <- rbind.data.frame(
```

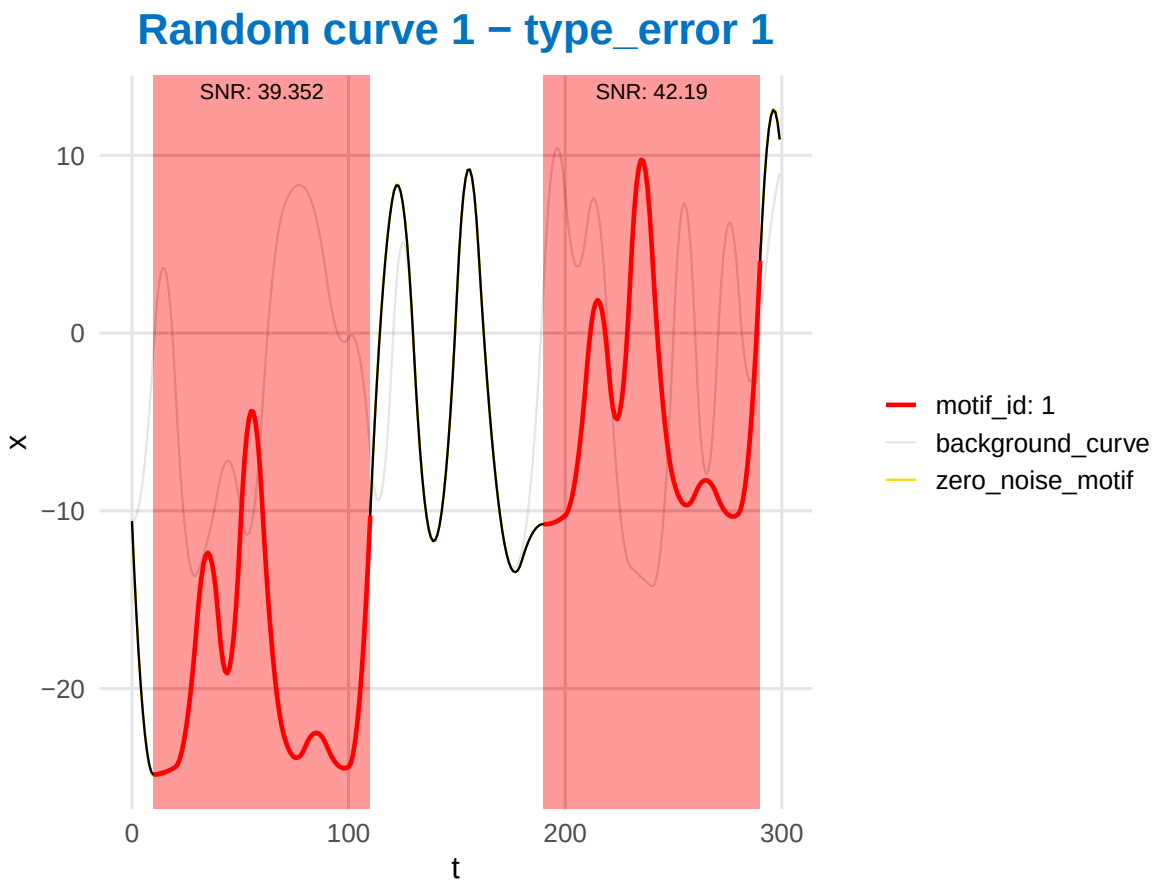
```
  c(1, 1, 20),
  c(2, 1, 2),
  c(1, 3, 1),
  c(1, 2, 1),
  c(1, 2, 15),
  c(1, 4, 1),
  c(2, 5, 1),
  c(2, 7, 1),
  c(2, 17, 1)
)

names(motif_str) <- c("motif_id", "curve", "start_break_pos")

mot1 <- list(
  "len" = mot_len,                # Length
  "coeffs" = NULL,               # Weights for the motif
  "occurrences" = motif_str %>% filter(motif_id == 1)
)

mot2 <- list(
  "len" = mot_len,
  "coeffs" = NULL,
  "occurrences" = motif_str %>% filter(motif_id == 2)
)

mot_details <- list(mot1, mot2)

# MATRIX NOISE
noise_str <- list(
  rbind(
    rep(2, 100),                       # Constant and identical
    c(rep(0.1, 50), rep(2, 50)),      # SD 0.1 first, SD 1 later
    c(rep(2, 50), rep(0.1, 50)),      # SD 1 first, 0.1 later
    c(seq(2, 0.1, len = 50), rep(0.1, 50))
  ),
  rbind(
    rep(0.0, 100),
    rep(0.5, 100),
    rep(1.0, 100),
    rep(5.0, 100)
  )
)

builder <- funMoDisco::motifSimulationBuilder(
  N = 20,
  len = 300,
  mot_details,
```

```
  distribution = 'beta'
)

curves <- funMoDisco::generateCurves(
  builder,
  noise_type = 'pointwise',
  noise_str = noise_str
)
#> [1] " --- Adding motifs to curve 1"
#> [1] " --- Adding motifs to curve 3"
#> [1] " --- Adding motifs to curve 2"
#> [1] " --- Adding motifs to curve 4"
#> [1] " --- Adding motifs to curve 5"
#> [1] " --- Adding motifs to curve 7"
#> [1] " --- Adding motifs to curve 17"

funMoDisco::plot_motifs(builder, curves, "plots_1")
#> pdf
#>   2
```

2) Set the motif position and add coeff noise

```
library(funMoDisco)

mot_len <- 100

# Struct specifying the motif ID, the number of curves, and the relative knot position
motif_str <- rbind.data.frame(
  c(1, 1, 20),
  c(1, 1, 2),
  c(1, 3, 1),
  c(1, 2, 1),
  c(1, 2, 15),
  c(1, 4, 1),
  c(1, 5, 1),
  c(1, 7, 1),
  c(2, 17, 1)
)

names(motif_str) <- c("motif_id", "curve", "start_break_pos")

mot1 <- list(
  "len" = mot_len,                # Length
  "coeffs" = NULL,               # Weights for the motif
  "occurrences" = motif_str %>% filter(motif_id == 1)
)

mot2 <- list(
  "len" = mot_len,
  "coeffs" = NULL,
  "occurrences" = motif_str %>% filter(motif_id == 2)
)

mot_details <- list(mot1, mot2)

# VECTOR NOISE
noise_str <- list(
  c(0.1, 1.0, 5.0),
  c(0.0, 0.0, 0.0)
)

builder <- funMoDisco::motifSimulationBuilder(
  N = 20,
  len = 300,
  mot_details,
  distribution = 'beta'
)
```

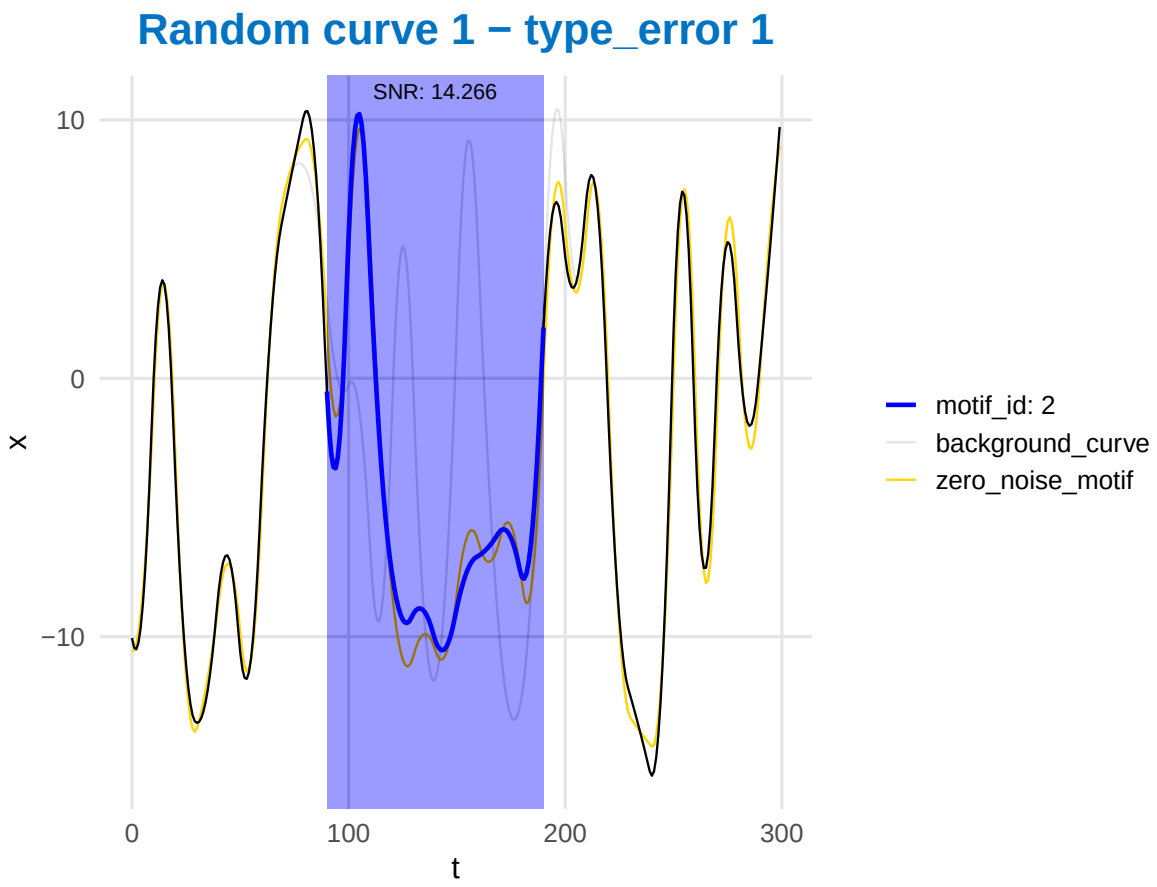
```r
curves <- funMoDisco::generateCurves(
  builder,
  noise_type = 'coeff',
  noise_str,
  only_der = FALSE
)
#> [1] " --- Adding motif 1 to curve 1 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 1 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 1 with noise  1"
#> [1] " --- Adding motif 1 to curve 1 with noise  1"
#> [1] " --- Adding motif 1 to curve 1 with noise  5"
#> [1] " --- Adding motif 1 to curve 1 with noise  5"
#> [1] " --- Adding motif 1 to curve 2 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 2 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 2 with noise  1"
#> [1] " --- Adding motif 1 to curve 2 with noise  1"
#> [1] " --- Adding motif 1 to curve 2 with noise  5"
#> [1] " --- Adding motif 1 to curve 2 with noise  5"
#> [1] " --- Adding motif 1 to curve 3 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 3 with noise  1"
#> [1] " --- Adding motif 1 to curve 3 with noise  5"
#> [1] " --- Adding motif 1 to curve 4 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 4 with noise  1"
#> [1] " --- Adding motif 1 to curve 4 with noise  5"
#> [1] " --- Adding motif 1 to curve 5 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 5 with noise  1"
#> [1] " --- Adding motif 1 to curve 5 with noise  5"
#> [1] " --- Adding motif 1 to curve 7 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 7 with noise  1"
#> [1] " --- Adding motif 1 to curve 7 with noise  5"
#> [1] " --- Adding motif 2 to curve 17 with noise  0"
#> [1] " --- Adding motif 2 to curve 17 with noise  0"
#> [1] " --- Adding motif 2 to curve 17 with noise  0"

funMoDisco::plot_motifs(builder, curves, "plots_2")
#> pdf
#>   2
```



3) Random motif position and add pointwse noise

```
library(funMoDisco)

mot_len <- 100

# Define motifs
mot1 <- list(
  "len" = mot_len,                # Length
  "coeffs" = NULL,                # Weights for the motif
  "occurrences" = 5
)

mot2 <- list(
  "len" = mot_len,
  "coeffs" = NULL,
  "occurrences" = 6
)
```

```r
mot_details <- list(mot1, mot2)

# Define noise structure
noise_str <- list(
  rbind(rep(2, 100)),
  rbind(rep(0.5, 100))
)

# Build motif simulation
builder <- funMoDisco::motifSimulationBuilder(
  N = 20,
  len = 300,
  mot_details,
  distribution = 'beta'
)

# Generate curves
curves <- funMoDisco::generateCurves(
  builder,
  noise_type = 'pointwise',
  noise_str,
  only_der = FALSE
)
#> [1] " --- Adding motifs to curve 8"
#> [1] " --- Adding motifs to curve 9"
#> [1] " --- Adding motifs to curve 16"
#> [1] " --- Adding motifs to curve 17"
#> [1] " --- Adding motifs to curve 1"
#> [1] " --- Adding motifs to curve 4"
#> [1] " --- Adding motifs to curve 6"
#> [1] " --- Adding motifs to curve 18"

# Plot motifs
funMoDisco::plot_motifs(builder, curves, "plots_3")
#> pdf
#>   2
```



4) Random motif position and add coeff noise

```r
library(funMoDisco)

mot_len <- 100

# Define motifs
mot1 <- list(
  "len" = mot_len,                # Length
  "weights" = NULL,               # Weights for the motif
  "occurrences" = 5
)

mot2 <- list(
  "len" = mot_len,
  "coeffs" = NULL,
  "occurrences" = 6
)
```

```r
mot_details <- list(mot1, mot2)

# Define vector noise
noise_str <- list(
  c(0.1, 5.0, 10.0),
  c(0.1, 5.0, 10.0)
)

# Build motif simulation
builder <- funMoDisco::motifSimulationBuilder(
  N = 20,
  len = 300,
  mot_details,
  distribution = 'beta'
)

# Generate curves
curves <- funMoDisco::generateCurves(
  builder,
  noise_type = 'coeff',
  noise_str,
  only_der = FALSE
)
#> [1] " --- Adding motif 1 to curve 3 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 3 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 3 with noise  5"
#> [1] " --- Adding motif 1 to curve 3 with noise  5"
#> [1] " --- Adding motif 1 to curve 3 with noise  10"
#> [1] " --- Adding motif 1 to curve 3 with noise  10"
#> [1] " --- Adding motif 2 to curve 4 with noise  0.1"
#> [1] " --- Adding motif 2 to curve 4 with noise  5"
#> [1] " --- Adding motif 2 to curve 4 with noise  10"
#> [1] " --- Adding motif 2 to curve 8 with noise  0.1"
#> [1] " --- Adding motif 2 to curve 8 with noise  0.1"
#> [1] " --- Adding motif 2 to curve 8 with noise  5"
#> [1] " --- Adding motif 2 to curve 8 with noise  5"
#> [1] " --- Adding motif 2 to curve 8 with noise  10"
#> [1] " --- Adding motif 2 to curve 8 with noise  10"
#> [1] " --- Adding motif 1 to curve 9 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 9 with noise  5"
#> [1] " --- Adding motif 1 to curve 9 with noise  10"
#> [1] " --- Adding motif 2 to curve 10 with noise  0.1"
#> [1] " --- Adding motif 2 to curve 10 with noise  0.1"
#> [1] " --- Adding motif 2 to curve 10 with noise  5"
#> [1] " --- Adding motif 2 to curve 10 with noise  5"
#> [1] " --- Adding motif 2 to curve 10 with noise  10"
#> [1] " --- Adding motif 2 to curve 10 with noise  10"
```

```
#> [1] " --- Adding motif 1 to curve 16 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 16 with noise  0.1"
#> [1] " --- Adding motif 1 to curve 16 with noise  5"
#> [1] " --- Adding motif 1 to curve 16 with noise  5"
#> [1] " --- Adding motif 1 to curve 16 with noise  10"
#> [1] " --- Adding motif 1 to curve 16 with noise  10"
#> [1] " --- Adding motif 2 to curve 20 with noise  0.1"
#> [1] " --- Adding motif 2 to curve 20 with noise  5"
#> [1] " --- Adding motif 2 to curve 20 with noise  10"

# Plot motifs
funMoDisco::plot_motifs(builder, curves, "plots_4")
#> pdf
#>   2
```

Additional functions

In addition to the functions previously described, the package includes a helper function that facilitates the direct transformation of the output from 'generateCurves' into a format suitable for

'discoverMotifs'. This function generates a comprehensive list that encompasses all curves, each containing the embedded patterns corresponding to various tested noise levels.

```
result <- funMoDisco::to_motifDiscovery(curves)
```

Additionally, a Shiny app is available, serving as a graphical user interface (GUI) that enables users to execute all the previously mentioned functions in a straightforward and intuitive manner. The app consistently provides summary plots, enhancing the user experience.

```
library(funMoDisco)

# Define motif structure
motif_str <- rbind.data.frame(
  c(1, 1, 20),
  c(1, 1, 2),
  c(1, 3, 1),
  c(1, 2, 1),
  c(1, 2, 15),
  c(1, 4, 1),
  c(1, 5, 1),
  c(1, 7, 1),
  c(2, 17, 1)
)

names(motif_str) <- c("motif_id", "curve", "start_break_pos")

# Define motifs
mot1 <- list(
  "len" = 100,                    # Length
  "weights" = NULL,               # Weights for the motif
  "appearance" = motif_str %>% filter(motif_id == 1)
)

mot2 <- list(
  "len" = 150,
  "weights" = NULL,
  "appearance" = motif_str %>% filter(motif_id == 2)
)

mot_details <- list(mot1, mot2)

# Define noise structure
noise_str <- list(
  rbind(rep(2, 100), c(rep(0.1, 50), rep(2, 50))),
  rbind(rep(0.0, 150), rep(5.0, 150))
)
```

```r
# Run motif simulation app
funMoDisco::motifSimulationApp(noise_str, mot_details)
```

Conclusion

The **discoverMotifs** function is a powerful tool for discovering functional motifs in complex datasets. With its flexibility, users can run multiple initializations, customize clustering parameters, simulate functional curves with motifs, and visualize the results in an intuitive way. Whether using **ProbKMA** or **funBIalign**, the funMoDisco package provides a robust solution for analyzing functional data and uncovering hidden patterns.