Package 'flacco'

July 22, 2025

Title Feature-Based Landscape Analysis of Continuous and Constrained Optimization Problems

Description Tools and features for ``Exploratory Landscape Analysis (ELA)'' of single-objective continuous optimization problems.
Those features are able to quantify rather complex properties, such as the global structure, separability, etc., of the optimization problems.

URL https://github.com/kerschke/flacco

BugReports https://github.com/kerschke/flacco/issues

License BSD_2_clause + file LICENSE

Encoding UTF-8

Depends R (>= 3.0.0)

Imports BBmisc, checkmate, mlr

Suggests devtools, e1071, ggplot2, lhs, MASS, Matrix, mda, mlbench, numDeriv, parallel, parallelMap, ParamHelpers, plotly, plyr, RANN, R.rsp, rpart, shape, shiny, smoof, testthat

LazyData yes

ByteCompile yes

Version 1.8

Date 2020-03-31

RoxygenNote 7.1.0

VignetteBuilder R.rsp

NeedsCompilation no

Author Pascal Kerschke [aut, cre],
Christian Hanster [ctb],
Jan Dagefoerde [ctb]

Maintainer Pascal Kerschke <kerschke@uni-muenster.de>

Repository CRAN

Date/Publication 2020-03-31 20:10:02 UTC

Contents

BBOBImport . . . 2
BBOBImportPage . . . 3
calculateFeatureSet . . . 3
computeGridCenters . . . 15
convertInitDesignToGrid . . . 16
createInitialSample . . . 17
featureList . . . 18
FeatureObject . . . 18
featureObject_sidebar . . . 20
FeatureSetCalculation . . . 20
FeatureSetCalculationComponent . . . 21
FeatureSetVisualization . . . 22
FeatureSetVisualizationComponent . . . 22
findLinearNeighbours . . . 23
findNearestPrototype . . . 24
functionInput . . . 25
ggplotFeatureImportance . . . 25
listAvailableFeatureSets . . . 28
measureTime . . . 29
plotBarrierTree2D . . . 29
plotBarrierTree3D . . . 31
plotCellMapping . . . 32
plotInformationContent . . . 34
runFlaccoGUI . . . 36
SmoofImport . . . 37
SmoofImportPage . . . 37

Index **38**

`BBOBImport` *Shiny server function for BBOB import page module*

Description

`BBOBImport` is a `shiny` server function which will control all aspects of the `BBOBImportPage` UI Module. It will be called with `callModule`.

Usage

```
BBOBImport(input, output, session, stringsAsFactors)
```

Arguments

input	[shiny-input] shiny input variable for the specific UI module.
output	[shiny-output object] shiny output variable for the specific UI module.
session	[shiny-session object] shiny session variable for the specific UI module.
stringsAsFactors	[logical(1)] How should strings be treated internally?

BBOBImportPage *Shiny UI-Module for Batch Import of BBOB Functions*

Description

BBOBImportPage is a shiny component which can be added to your shiny app so that you get a batch import for several BBOB functions.

Usage

```
BBOBImportPage(id)
```

Arguments

id	[character(1)] Character representing the namespace of the shiny component.

Details

It will load a CSV-file with BBOB parameters (the function ID, instance ID and problem dimension) and then calculate the selected features for the specific function(s).

---

calculateFeatureSet *Calculate Landscape Features*

Description

Performs an Exploratory Landscape Analysis of a continuous function and computes various features, which quantify the function's landscape. Currently, the following feature sets are provided:

- CM: cell mapping features (`"cm_angle"`, `"cm_conv"`, `"cm_grad"`)
- ELA: classical ELA features (`"ela_conv"`, `"ela_curv"`, `"ela_distr"`, `"ela_level"`, `"ela_local"`, `"ela_meta"`)
- GCM: general cell mapping features (`"gcm"`)
- BT: barrier tree features (`"bt"`)
- IC: information content features (`"ic"`)
- Basic: basic features (`"basic"`)
- Disp: dispersion features (`"disp"`)
- LiMo: linear model features (`"limo"`)
- NBC: nearest better clustering features (`"nbc"`)
- PC: principal component features (`"pca"`)

Usage

```
calculateFeatureSet(feat.object, set, control, ...)

calculateFeatures(feat.object, control, ...)
```

Arguments

`feat.object`	[`FeatureObject`] A feature object as created by `createFeatureObject`.
`set`	[`character(1)`] Name of the feature set, which should be computed. All possible feature sets can be listed using `listAvailableFeatureSets`.
`control`	[`list`] A `list`, which stores additional control arguments. For further information, see details.
`...`	[any] Further arguments, e.g. handled by `optim` (within the computation of the ELA local search features) or `density` (within the computation of the ELA y-distribution features).

Details

Note that if you want to speed up the runtime of the features, you might consider running your feature computation parallelized. For more information, please refer to the `parallelMap` package or to `https://mlr.mlr-org.com/articles/tutorial/parallelization.html`.

Furthermore, please consider adapting the feature computation to your needs. Possible `control` arguments are:

- general:
  - show_progress: Show progress bar when computing the features? The default is TRUE.
  - subset: Specify a subset of features that should be computed. Per default, all features will be computed.
  - allow_cellmapping: Should cell mapping features be computed? The default is TRUE.
  - allow_costs: Should expensive features, i.e. features, which require additional function evaluations, be computed? The default is TRUE if the feature object provides a function, otherwise FALSE.
  - blacklist: Which features should NOT be computed? The default is NULL, i.e. none of the features will be excluded.
- cell mapping angle features:
  - cm_angle.show_warnings: Should possible warnings about NAs in the feature computation be shown? The default is FALSE.
- cell mapping convexity features:
  - cm_conv.diag: Should cells, which are located on the diagonal compared to the current cell, be considered as neighbouring cells? The default is FALSE, i.e. only cells along the axes are considered as neighbours.
  - cm_conv.dist_method: Which distance method should be used for computing the distance between two observations? All methods of dist are possible options with "euclidean" being the default.
  - cm_conv.minkowski_p: Value of p in case dist_meth is "minkowski". The default is 2, i.e. the euclidean distance.
  - cm_conv.fast_k: Percentage of elements that should be considered within the nearest neighbour computation. The default is 0.05.
- cell mapping gradient homogeneity features:
  - cm_grad.dist_tie_breaker: How will ties be broken when different observations have the same distance to an observation? Possible values are "sample", "first" and "last". The default is "sample".
  - cm_grad.dist_method: Which distance method should be used for computing the distance between two observations? All methods of dist are possible options with "euclidean" being the default.
  - cm_grad.minkowski_p: Value of p in case dist_meth is "minkowski". The default is 2, i.e. the euclidean distance.
  - cm_grad.show_warnings: Should possible warnings about (almost) empty cells be shown? The default is FALSE.
- ELA convexity features:
  - ela_conv.nsample: Number of samples that are drawn for calculating the convexity features. The default is 1000.
  - ela_conv.threshold: Threshold of the linearity, i.e. the tolerance to / deviation from perfect linearity, in order to still be considered linear. The default is 1e-10.
- ELA curvature features:
  - ela_curv.sample_size: Number of samples used for calculating the curvature features. The default is 100*d.

  - ela_curv.{delta, eps, zero_tol, r, v}: Parameters used by grad and hessian within the approximation of the gradient and hessian. The default values are identical to the ones from the corresponding functions. Note that we slightly modified hessian in order to assure that we do not exceed the boundaries during the estimation of the Hessian.
- ELA distribution features:
  - ela_distr.smoothing_bandwidth: The smoothing bandwidth, which should be used within the density estimation. The default is "SJ".
  - ela_distr.modemass_threshold: Threshold that is used in order to classify whether a minimum can be considered as a peak. The default is 0.01.
  - ela_distr.skewness_type: Algorithm type for computing the skewness. The default is 3.
  - ela_distr.kurtosis_type: Algorithm type for computing the kurtosis. The default is 3.
- ELA levelset features:
  - ela_level.quantiles: Cutpoints (quantiles of the objective values) for splitting the objective space. The default is c(0.10, 0.25, 0.50).
  - ela_level.classif_methods: Methods for classifying the artificially splitted objective space. The default is c("lda", "qda", "mda").
  - ela_level.resample_method: Resample technique for training the model, cf. ResampleDesc. The default is "CV".
  - ela_level.resample_iterations: Number of iterations of the resampling method. The default is 10.
  - ela_level.resample_info: Should information regarding the resampling be printed? The default is FALSE.
  - ela_level.parallelize: Should the levelset features be computed in parallel? The default is FALSE.
  - ela_level.parallel.mode: Which mode should be used for the parallelized computation? Possible options are "local", "multicore", "socket" (default), "mpi" and "BatchJobs". Note that in case you are using a windows computer you can only use the "socket" mode.
  - ela_level.parallel.cpus: On how many cpus do you want to compute the features in parallel? Per default, all available cpus are used.
  - ela_level.parallel.level: On which level should the parallel computation be performed? The default is "mlr.resample", i.e. the internal resampling (performed using mlr) will be done in parallel.
  - ela_level.parallel.logging: Should slave output be logged? The default is FALSE.
  - ela_level.parallel.show_info: Should verbose output of function calls be printed on the console? The default is FALSE.
- ELA local search features:
  - ela_local.local_searches: Number of local searches. The default is 50 * d with d being the number of features (i.e. the dimension).
  - ela_local.optim_method: Local search algorithm. The default is "L-BFGS-B".
  - ela_local.optim.{lower, upper}: Lower and upper bounds to be considered by the local search algorithm. Per default, the boundaries are the same as defined within the feature object (in case of "L-BFGS-B") or infinity (for all others).

  - ela_local.optim_method_control: Control settings of the local search algorithm. The default is an empty list.
  - ela_local.sample_seed: Seed, which will be set before the selection of the initial start points of the local search. The default is sample(1:1e6, 1).
  - ela_local.clust_method: Once the local searches converge, basins have to be assigned. This is done using hierarchical clustering methods from hclust. The default is "single", i.e. *single linkage clustering*.
  - ela_local.clust_cut_function: A function of a hierarchical clustering cl, which defines at which height the dendrogramm should be splitted into clusters (cf. cutree). The default is function(cl) as.numeric(quantile(cl$height, 0.1)), i.e. the 10%-quantile of all the distances between clusters.
- GCM features:
  - gcm.approaches: Which approach(es) should be used when computing the representatives of a cell. The default are all three approaches, i.e. c("min", "mean", "near").
  - gcm.cf_power: Theoretically, we need to compute the canonical form to the power of infinity. However, we use this value as approximation of infinity. The default is 256.
- barrier tree features:
  - gcm.approaches: Which approach(es) should be used when computing the representatives of a cell. The default are all three approaches, i.e. c("min", "mean", "near").
  - gcm.cf_power: Theoretically, we need to compute the canonical form to the power of infinity. However, we use this value as approximation of infinity. The default is 256.
  - bt.base: Maximum number of basins, which are joined at a single breakpoint. The default is 4L.
  - bt.max_depth: Maximum number of levels of the barrier tree. The default is 16L.
- information content features:
  - ic.epsilon: Epsilon values as described in section V.A of Munoz et al. (2015). The default is c(0, 10^(seq(-5, 15, length.out = 1000)).
  - ic.sorting: Sorting strategy, which is used to define the tour through the landscape. Possible values are "nn" (= default) and "random".
  - ic.sample.generate: Should the initial design be created using a LHS? The default is FALSE, i.e. the initial design from the feature object will be used.
  - ic.sample.dimensions: Dimensions of the initial design, if created using a LHS. The default is feat.object$dimension.
  - ic.sample.size: Size of the initial design, if created using a LHS. The default is 100 * feat.object$dimension.
  - ic.sample.lower: Lower bounds of the initial design, if created with a LHS. The default is 100 * feat.object$lower.
  - ic.sample.upper: Upper bounds of the initial design, if created with a LHS. The default is 100 * feat.object$upper.
  - ic.aggregate_duplicated: How should observations, which have duplicates in the decision space, be aggregated? The default is mean.
  - ic.show_warnings: Should warnings be shown, when possible duplicates are removed? The default is FALSE.
  - ic.seed: Possible seed, which can be used for making your experiments reproducable. Per default, a random number will be drawn as seed.

  - ic.nn.start: Which observation should be used as starting value, when exploring the landscape with the nearest neighbour approach. The default is a randomly chosen integer value.
  - ic.nn.neighborhood: In order to provide a fast computation of the features, we use RANN::nn2 for computing the nearest neighbors of an observation. Per default, we consider the 20L closest neighbors for finding the nearest not-yet-visited observation. If all of those neighbors have been visited already, we compute the distances to the remaining points separately.
  - ic.settling_sensitivity: Threshold, which should be used for computing the "settling sensitivity". The default is 0.05 (as used in the corresponding paper).
  - ic.info_sensitivity: Portion of partial information sensitivity. The default is 0.5 (as used in the paper).

- dispersion features:
  - disp.quantiles: Quantiles, which should be used for defining the "best" elements of the entire initial design. The default is c(0.02, 0.05, 0.1, 0.25).
  - disp.dist_method: Which distance method should be used for computing the distance between two observations? All methods of dist are possible options with "euclidean" being the default.
  - disp.minkowski_p: Value of p in case dist_meth is "minkowski". The default is 2, i.e. the euclidean distance.

- nearest better clustering features:
  - nbc.dist_method: Which distance method should be used for computing the distance between two observations? All methods of dist are possible options with "euclidean" being the default.
  - nbc.minkowski_p: Value of p in case dist_meth is "minkowski". The default is 2, i.e. the euclidean distance.
  - nbc.dist_tie_breaker: How will ties be broken when different observations have the same distance to an observation? Possible values are "sample", "first" and "last". The default is "sample".
  - nbc.cor_na: How should NA's be handled when computing correlations? Any method from the argument use of the function cor is possible. The default is "pairwise.complete.obs".
  - nbc.fast_k: In case of euclidean distances, the method can find neighbours faster. This parameter controls the percentage of observations that should be considered when looking for the nearest better neighbour, i.e. the nearest neighbour with a better objective value. The default is 0.05, i.e. the 5

- principal component features:
  - pca.{cov, cor}_{x, init}: Which proportion of the variance should be explained by the principal components given a principal component analysis based on the covariance / correlation matrix of the decision space (x) or the entire initial design (init)? The defaults are 0.9.

Value

list of (numeric) features:

- cm_angle – angle features (10):
  These features are based on the location of the worst and best element within each cell. To be precise, their distance to the cell center and the angle between these three elements (at the center) are the foundation:
  - dist_ctr2{best, worst}.{mean, sd}: arithmetic mean and standard deviation of distances from the cell center to the best / worst observation within the cell (over all cells)
  - angle.{mean, sd}: arithmetic mean and standard deviation of angles (in degree) between worst, center and best element of a cell (over all cells)
  - y_ratio_best2worst.{mean, sd}: arithmetic mean and standard deviation of the ratios between the distance of the worst and best element within a cell and the worst and best element in the entire initial design (over all cells);
    note that the distances are only measured in the objective space
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features
- cm_conv – cell mapping convexity features (6):
  Each cell will be represented by an observation (of the initial design), which is located closest to the cell center. Then, the objectives of three neighbouring cells are compared:
  - {convex, concave}.hard: if the objective of the inner cell is above / below the two outer cells, there is strong evidence for convexity / concavity
  - {convex, concave}.soft: if the objective of the inner cell is above / below the arithmetic mean of the two outer cells, there is weak evidence for convexity / concavity
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features
- cm_grad – gradient homogeneity features (4):
  Within a cell of the initial grid, the gradients between each observation and its nearest neighbour observation are computed. Those gradients are then directed towards the smaller of the two objective values and afterwards normalized. Then, the length of the sum of all the directed and normalized gradients within a cell is computed. Based on those measurements (one per cell) the following features are computed:
  - {mean, sd}: arithmetic mean and standard deviation of the aforementioned lengths
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features
- ela_conv – ELA convexity features (6):
  Two observations are chosen randomly from the initial design. Then, a linear (convex) combination of those observations is calculated – based on a random weight from [0, 1]. The corresponding objective value will be compared to the linear combination of the objectives from the two original observations. This process is replicated convex.nsample (per default 1000) times and will then be aggregated:
  - {convex_p, linear_p}: percentage of convexity / linearity
  - linear_dev.{orig, abs}: average (original / absolute) deviation between the linear combination of the objectives and the objective of the linear combination of the observations

  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- ela_curv – ELA curvature features (26):
  Given a feature object, curv.sample_size samples (per default 100 * d with d being the number of features) are randomly chosen. Then, the gradient and hessian of the function are estimated based on those points and the following features are computed:

  - grad_norm.{min, lq, mean, median, uq, max, sd, nas}: aggregations (minimum, lower quartile, arithmetic mean, median, upper quartile, maximum, standard deviation and percentage of NAs) of the gradients' lengths
  - grad_scale.{min, lq, mean, median, uq, max, sd, nas}: aggregations of the ratios between biggest and smallest (absolute) gradient directions
  - hessian_cond.{min, lq, mean, median, uq, max, sd, nas}: aggregations of the ratios of biggest and smallest eigenvalue of the hessian matrices
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- ela_distr – ELA y-distribution features (5):

  - skewness: skewness of the objective values
  - kurtosis: kurtosis of the objective values
  - number_of_peaks: number of peaks based on an estimation of the density of the objective values
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- ela_level – ELA levelset features (20):

  - mmce_{methods}_{quantiles}: mean misclassification error of each pair of classification method and quantile
  - {method1}_{method2}_{quantiles}: ratio of all pairs of classification methods for all quantiles
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- ela_local – ELA local search features (16):
  Based on some randomly chosen points from the initial design, a pre-defined number of local searches (ela_local.local_searches) are executed. Their optima are then clustered (using hierarchical clustering), assuming that local optima that are located close to each other, likely belong to the same basin. Given those basins, the following features are computed:

  - n_loc_opt.{abs, rel}: the absolute / relative amount of local optima
  - best2mean_contr.orig: each cluster is represented by its center; this feature is the ratio of the objective values of the best and average cluster
  - best2mean_contr.ratio: each cluster is represented by its center; this feature is the ratio of the differences in the objective values of average to best and worst to best cluster
  - basin_sizes.avg_{best, non_best, worst}: average basin size of the best / non-best / worst cluster(s)

  - fun_evals.{min, lq, mean, median, uq, max, sd}: aggregations of the performed local searches
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features
- ela_meta – ELA meta model features (11):
  Given an initial design, linear and quadratic models of the form objective ~ features are created. Both versions are created with and without simple interactions (e.g., x1:x2). Based on those models, the following features are computed:
  - lin_simple.{adj_r2, intercept}: adjusted R^2 (i.e. model fit) and intercept of a simple linear model
  - lin_simple.coef.{min, max, max_by_min}: smallest and biggest (non-intercept) absolute coefficients of the simple linear model, and their ratio
  - {lin_w_interact, quad_simple, quad_w_interact}.adj_r2: adjusted R^2 (i.e. the model fit) of a linear model with interactions, and a quadratic model with and without interactions
  - quad_simple.cond: condition of a simple quadratic model (without interactions), i.e. the ratio of its (absolute) biggest and smallest coefficients
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features
- gcm – general cell mapping (GCM) features (75):
  Computes general cell mapping features based on the Generalized Cell Mapping (GCM) approach, which interpretes the cells as absorbing Markov chains. Computations are performed based on three different approaches: taking the best (min) or average (mean) objective value of a cell or the closest observation (near) to a cell as representative. For each of these approaches the following 25 features are computed:
  - attractors, pcells, tcells, uncertain: relative amount of attractor, periodic, transient and uncertain cells
  - basin_prob.{min, mean, median, max, sd}: aggregations of the probabilities of each basin of attraction
  - basin_certain.{min, mean, median, max, sd}: aggregations of the (relative) size of each basin of attraction, in case only certain cells are considered (i.e. cells, which only point towards one attractor)
  - basin_uncertain.{min, mean, median, max, sd}: aggregations of the (relative) size of each basin of attraction, in case uncertain cells are considered (i.e. a cell, which points to multiple attractors contributes to each of its basins)
  - best_attr.{prob, no}: probability of finding the attractor with the best objective value and the (relative) amount of those attractors (i.e. the ratio of the number of attractors with the best objective value and the total amount of cells)
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features
- bt – barrier tree features (90):
  Computes barrier tree features, based on a Generalized Cell Mapping (GCM) approach. Computations are performed based on three different approaches: taking the best (min) or average (mean) objective value of a cell or the closest observation (near) to a cell as representative.

For each of these approaches the following 31 features are computed:

  - `levels`: absolute number of levels of the barrier tree
  - `leaves`: absolute number of leaves (i.e. local optima) of the barrier tree
  - `depth`: range between highest and lowest node of the tree
  - `depth_levels_ratio`: ratio of depth and levels
  - `levels_nodes_ratio`: ratio of number of levels and number of (non-root) nodes of the tree
  - `diffs.{min, mean, median, max, sd}`: aggregations of the height differences between a node and its predecessor
  - `level_diffs.{min, mean, median, max, sd}`: aggregations of the average height differences per level
  - `attractor_dists.{min, mean, median, max, sd}`: aggregations of the (euclidean) distances between the local and global best cells (attractors)
  - `basin_ratio.{uncertain, certain, most_likely}`: ratios of maximum and minimum size of the basins of attractions; here, a cell might belong to different attractors (uncertain), exactly one attractor (certain) or the attractor with the highest probability
  - `basin_intersection.{min, mean, median, max, sd}`: aggregations of the intersection between the basin of the global best value and the basins of all local best values
  - `basin_range`: range of a basin (euclidean distance of widest range per dimension)
  - `costs_{fun_evals, runtime}`: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- `ic` – information content features (7):
  Computes features based on the Information Content of Fitness Sequences (ICoFiS) approach (cf. Munoz et al., 2015). In this approach, the information content of a continuous landscape, i.e. smoothness, ruggedness, or neutrality, are quantified. While common analysis methods were able to calculate the information content of discrete landscapes, the ICoFiS approach provides an adaptation to continuous landscapes that accounts e.g. for variable step sizes in random walk sampling:

  - `h.max`: "maximum information content" (entropy) of the fitness sequence, cf. equation (5)
  - `eps.s`: "settling sensitivity", indicating the epsilon for which the sequence nearly consists of zeros only, cf. equation (6)
  - `eps.max`: similar to `eps.s`, but in contrast to the former `eps.max` guarantees non-missing values; this simply is the epsilon-value for which H(`eps.max`) == `h.max`
  - `eps.ratio`: "ratio of partial information sensitivity", cf. equation (8), where the ratio is `0.5`
  - `m0`: "initial partial information", cf. equation (7)
  - `costs_{fun_evals, runtime}`: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- `basic` – basic features (15):
  Very simple features, which can be read from the feature object (without any computational efforts):

  - {dim, observations}: number of features / dimensions and observations within the initial sample
  - {lower, upper, objective, blocks}_{min, max}: minimum and maximum value of all lower and upper bounds, the objective values and the number of blocks / cells (per dimension)
  - cells_{filled, total}: number of filled (i.e. non-empty) cells and total number of cells
  - {minimize_fun}: logical value, indicating whether the optimization function should be minimized
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- disp – dispersion features (18):
  Computes features based on the comparison of the dispersion of pairwise distances among the 'best' elements and the entire initial design:

  - {ratio, diff}_{mean, median}_{02, 05, 10, 25}: ratio and difference of the mean / median distances of the distances of the 'best' objectives vs. 'all' objectives
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- limo – linear model features (14):
  Linear models are computed per cell, provided the decision space is divided into a grid of cells. Each one of the models has the form objective ~ features.

  - avg_length.{reg, norm}: length of the average coefficient vector (based on regular and normalized vectors)
  - length_{mean, sd}: arithmetic mean and standard deviation of the lengths of all coefficient vectors
  - cor.{reg, norm}: correlation of all coefficient vectors (based on regular and normalized vectors)
  - ratio_{mean, sd}: arithmetic mean and standard deviation of the ratios of (absolute) maximum and minimum (non-intercept) coefficients per cell
  - sd_{ratio, mean}.{reg, norm}: max-by-min-ratio and arithmetic mean of the standard deviations of the (non-intercept) coefficients (based on regular and normalized vectors)
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

- nbc – nearest better (clustering) features (7):
  Computes features based on the comparison of nearest neighbour and nearest better neighbour, i.e., the nearest neighbor with a better performance / objective value value.

  - nn_nb.{sd, mean}_ratio: ratio of standard deviations and arithmetic mean based on the distances among the nearest neighbours and the nearest better neighbours
  - nn_nb.cor: correlation between distances of the nearest neighbours and the distances of the nearest better neighbours
  - dist_ratio.coeff_var: coefficient of variation of the distance ratios

  - nb_fitness.cor: correlation between fitness value and count of observations to whom the current observation is the nearest better neighbour (the so-called "indegree").
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features
- pca – principal component (analysis) features (10):
  - expl_var.{cov, cor}_{x, init}: proportion of the explained variance when applying PCA to the covariance / correlation matrix of the decision space (x) or the entire initial design (init)
  - expl_var_PC1.{cov, cor}_{x, init}: proportion of variance, which is explained by the first principal component – when applying PCA to the covariance / correlation matrix of the decision space (x) or the entire initial design
  - costs_{fun_evals, runtime}: number of (additional) function evaluations and runtime (in seconds), which were needed for the computation of these features

References

- Kerschke, P., and Trautmann, H. (2019): "Comprehensive Feature-Based Landscape Analysis of Continuous and Constrained Optimization Problems Using the R-package flacco", in: Applications in Statistical Computing – From Music Data Analysis to Industrial Quality Improvement, pp. 93-123, Springer. (https://link.springer.com/chapter/10.1007/978-3-030-25147-5_7).
- Kerschke, P., Preuss, M., Hernandez, C., Schuetze, O., Sun, J.-Q., Grimme, C., Rudolph, G., Bischl, B., and Trautmann, H. (2014): "Cell Mapping Techniques for Exploratory Landscape Analysis", in: EVOLVE – A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V, pp. 115-131 (http://dx.doi.org/10.1007/978-3-319-07494-8_9).
- Kerschke, P., Preuss, M., Wessing, S., and Trautmann, H. (2015): "Detecting Funnel Structures by Means of Exploratory Landscape Analysis", in: Proceedings of the 17th Annual Conference on Genetic and Evolutionary Computation (GECCO '15), pp. 265-272 (http://dx.doi.org/10.1145/2739480.2754642).
- Lunacek, M., and Whitley, D. (2006): "The dispersion metric and the CMA evolution strategy", in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06), pp. 477-484 (http://dx.doi.org/10.1145/1143997.1144085).
- Mersmann, O., Bischl, B., Trautmann, H., Preuss, M., Weihs, C., and Rudolph, G. (2011): "Exploratory Landscape Analysis", in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO '11), pp. 829-836 (http://dx.doi.org/10.1145/2001576.2001690).
- Munoz, M. A., Kirley, M., and Halgamuge, S. K. (2015): "Exploratory Landscape Analysis of Continuous Space Optimization Problems Using Information Content", in: IEEE Transactions on Evolutionary Computation (19:1), pp. 74-87 (http://dx.doi.org/10.1109/TEVC.2014.2302006).

Examples

```
# (1) create a feature object:
X = t(replicate(n = 2000, expr = runif(n = 5, min = -10, max = 10)))
```

```
## Not run: feat.object = createFeatureObject(X = X, fun = function(x) sum(x^2))

# (2) compute all non-cellmapping features
ctrl = list(allow_cellmapping = FALSE)
## Not run: features = calculateFeatures(feat.object, control = ctrl)

# (3) in order to allow the computation of the cell mapping features, one
# has to provide a feature object that has knowledge about the number of
# cells per dimension:
f = function(x) sum(x^2)
feat.object = createFeatureObject(X = X, fun = f, blocks = 3)
## Not run: features = calculateFeatures(feat.object)

# (4) if you want to compute a specific feature set, you can use
# calculateFeatureSet:
features.angle = calculateFeatureSet(feat.object, "cm_angle")

# (5) as noted in the details, it might be useful to compute the levelset
# features parallelized:
## Not run:
library(parallelMap)
library(parallel)
n.cores = detectCores()
parallelStart(mode = "socket", cpus = n.cores,
  logging = FALSE, show.info = FALSE)
system.time((levelset.par = calculateFeatureSet(feat.object, "ela_level")))
parallelStop()
system.time((levelset.seq = calculateFeatureSet(feat.object, "ela_level")))
## End(Not run)
```

`computeGridCenters` *Compute the Cell Centers of a Cell Mapping Grid*

Description

Computes the cell centers and the corresponding cell IDs of a cell mapping grid.

Usage

```
computeGridCenters(lower, upper, blocks)
```

Arguments

`lower`	[`numeric` or `integer`] The lower limits per dimension.
`upper`	[`numeric` or `integer`] The upper limits per dimension.
`blocks`	[`integer`] The number of blocks per dimension.

Value

[data.frame].
A data.frame, which includes the coordinates of the cell centers, as well as the corresponding cell ID (cell.ID).

Examples

```
computeGridCenters(lower = -10, upper = 10, blocks = c(10, 5, 8))
```

---

convertInitDesignToGrid

Converts an Initial Design into a Cell Mapping Grid

Description

This function takes an initial design – with rows being the observations and columns standing for the dimensions (plus the corresponding objective) – and adds an additional column to the data.frame. This additional column states the cell ID for each observation.

Usage

```
convertInitDesignToGrid(init, lower, upper, blocks)
```

Arguments

init	[data.frame] The initial design, consisting of d + 1 columns (d dimensions and one column for the objective value) and one row per observation.
lower	[numeric or integer] The lower limits per dimension.
upper	[numeric or integer] The upper limits per dimension.
blocks	[integer] The number of blocks per dimension.

Value

[data.frame].
A data.frame, which includes an additional column (cell.ID) compared to the initial design (init). The cell.ID will be a value between 1 and prod(blocks).

Examples

```
# (1) create an initial design:
X = t(replicate(n = 200, expr = runif(n = 5, min = -10, max = 10)))
f = function(x) sum(x^2)
y = apply(X = X, MARGIN = 1, FUN = f)
init = data.frame(X, y = y)

# (2) compute the cell mapping grid
convertInitDesignToGrid(init = init, lower = -10, upper = 10, blocks = 20)
```

---

createInitialSample *Create Initial Sample*

Description

Convenient helper function, which creates an initial sample - either based on random (uniform) sampling or using latin hypercube sampling.

Usage

```
createInitialSample(n.obs, dim, control)
```

Arguments

n.obs	[integer(1)] Number of observations.
dim	[integer(1)] Number of dimensions.
control	[list] Control argument. For further information refer to the details.

Details

Per default, this function will produce `n.obs` observations of size `dim` in the range from 0 to 1. If you want to create a more specific initial sample, the following control arguments might be helpful:

- `init_sample.type`: Should the initial sample be created based on random uniform sampling (`"random"`) or on a latin hypercube sample (`"lhs"`)? The default is `"random"`.
- `init_sample.lower`: The lower bounds of the initial sample. Either a vector of size `dim` or a scalar (if all lower bounds are identical). The default is `0`.
- `init_sample.upper`: The upper bounds of the initial sample. Either a vector of size `dim` or a scalar (if all upper bounds are identical). The default is `1`.

Value

[matrix].
A matrix, consisting of `n.obs` rows of `dim`-dimensional observations.

Examples

```
# (1) create a simple initial sample:
X = createInitialSample(300, 5)
summary(X)

# (2) create a more specific initial sample:
ctrl = list(init_sample.type = "lhs",
  init_sample.lower = c(-5, 2, 0),
  init_sample.upper = 10)
X = createInitialSample(200, 3, control = ctrl)
summary(X)
```

`featureList` *Feature List*

Description

Contains a list of features. This could be the result of a feature selection (based on a nested resampling strategy) executed on the `Glass` data.

`FeatureObject` *Create a Feature Object*

Description

Create a `FeatureObject`, which will be used as input for all the feature computations.

Usage

```
createFeatureObject(
  init,
  X,
  y,
  fun,
  minimize,
  lower,
  upper,
  blocks,
  objective,
  force = FALSE
)
```

Arguments

init	[data.frame] A data.frame, which can be used as initial design. If not provided, it will be created either based on the initial sample X and the objective values y or X and the function definition fun.
X	[data.frame or matrix] A data.frame or matrix containing the initial sample. If not provided, it will be extracted from init.
y	[numeric or integer] A vector containing the objective values of the initial design. If not provided, it will be extracted from init.
fun	[function] A function, which allows the computation of the objective values. If it is not provided, features that require additional function evaluations, can't be computed.
minimize	[logical(1)] Should the objective function be minimized? The default is TRUE.
lower	[numeric or integer] The lower limits per dimension.
upper	[numeric or integer] The upper limits per dimension.
blocks	[integer] The number of blocks per dimension.
objective	[character(1)] The name of the feature, which contains the objective values. The default is "y".
force	[logical(1)] Only change this parameter IF YOU KNOW WHAT YOU ARE DOING! Per default (force = FALSE), the function checks whether the total number of cells that you are trying to generate, is below the (hard-coded) internal maximum of 25,000 cells. If you set this parameter to TRUE, you agree that you want to exceed that internal limit. Note: *Exploratory Landscape Analysis (ELA)* is only useful when you are limited to a small budget (i.e., a small number of function evaluations) and in such scenarios, the number of cells should also be kept low!

Value

[FeatureObject].

Examples

```
# (1a) create a feature object using X and y:
X = createInitialSample(n.obs = 500, dim = 3,
  control = list(init_sample.lower = -10, init_sample.upper = 10))
y = apply(X, 1, function(x) sum(x^2))
feat.object1 = createFeatureObject(X = X, y = y,
  lower = -10, upper = 10, blocks = c(5, 10, 4))
```

```
# (1b) create a feature object using X and fun:
feat.object2 = createFeatureObject(X = X,
  fun = function(x) sum(sin(x) * x^2),
  lower = -10, upper = 10, blocks = c(5, 10, 4))

# (1c) create a feature object using a data.frame:
feat.object3 = createFeatureObject(iris[,-5], blocks = 5,
  objective = "Petal.Length")

# (2) have a look at the feature objects:
feat.object1
feat.object2
feat.object3

# (3) now, one could calculate features
calculateFeatureSet(feat.object1, "ela_meta")
calculateFeatureSet(feat.object2, "cm_grad")
library(plyr)
calculateFeatureSet(feat.object3, "cm_angle", control = list(cm_angle.show_warnings = FALSE))
```

---

featureObject_sidebar *Shiny UI-Module for Function Input*

Description

featObject_sidebar is a shiny UI-component which can be added to your shiny app so that you can easily generate a feature object by providing all relevant information.

Usage

```
featureObject_sidebar(id)
```

Arguments

id	[character(1)] Character representing the namespace of the shiny component.

FeatureSetCalculation *Shiny Server Function for Feature Set Component*

Description

FeatureSetCalculation is a shiny server function which will control all aspects of the FeatureSetCalculationComponen
UI Module. Will be called with callModule.

Usage

```
FeatureSetCalculation(input, output, session, stringsAsFactors, feat.object)
```

Arguments

input	[shiny-input] shiny input variable for the specific UI module.
output	[shiny-output object] shiny output variable for the specific UI module.
session	[shiny-session object] shiny session variable for the specific UI module.
stringsAsFactors	[logical(1)] How should strings be treated internally?
feat.object	[FeatureObject] A feature object as created by createFeatureObject.

Details

It will take the user input and calculate the selected feature set. In order to calculate a feature set, the function needs a FeatureObject.

FeatureSetCalculationComponent

Shiny UI-Module for Calculating and Displaying Feature Sets

Description

FeatureSetCalculationComponent is a shiny UI-component which can be added to your shiny app so that you can calculate and display different feature sets.

Usage

```
FeatureSetCalculationComponent(id)
```

Arguments

id	[character(1)] Character representing the namespace of the shiny component.

Details

The component integrates a select-Input for choosing the feature set, which should be calculated and displayed in a table. With the download button the calculated features can be exported as CSV-file.

`FeatureSetVisualization`

Shiny Server Function for Feature Set Component

Description

`FeatureSetVisualization` is a `shiny` server function which will control all aspects of the `FeatureSetVisualizationComp` UI-Module. It will be called with `callModule`.

Usage

```
FeatureSetVisualization(input, output, session, stringsAsFactors, feat.object)
```

Arguments

`input`	[`shiny-input`] `shiny` input variable for the specific UI module.
`output`	[`shiny-output object`] `shiny` output variable for the specific UI module.
`session`	[`shiny-session object`] `shiny` session variable for the specific UI module.
`stringsAsFactors`	[`logical(1)`] How should strings be treated internally?
`feat.object`	[`FeatureObject`] A feature object as created by `createFeatureObject`.

Details

It will take the user input and plot the selected visualization. To create a flacco plot, the function needs a `FeatureObject`.

`FeatureSetVisualizationComponent`

Shiny Component for Visualizing the Feature Sets

Description

`FeatureSetVisualizationComponent` is a `shiny` component which can be added to your `shiny` app so that you can display different feature set plots.

Usage

```
FeatureSetVisualizationComponent(id)
```

Arguments

id	[character(1)] Character representing the namespace of the shiny component.

Details

It integrates a select input where the user can select the plot which should be created.

---

findLinearNeighbours *Find Neighbouring Cells*

Description

Given a vector of cell IDs (cell.ids) and a vector (blocks), which defines the number of blocks / cells per dimension, a list of all combinations of (linearly) neighbouring cells around each element of cell.ids is returned.

Usage

```
findLinearNeighbours(cell.ids, blocks, diag = FALSE)
```

Arguments

cell.ids	[integer] Vector of cell IDs (one number per cell) for which the neighbouring cells should be computed.
blocks	[integer] The number of blocks per dimension.
diag	[logical(1)] logical, indicating whether only cells that are located parallel to the axes should be considered (diag = FALSE) as neighbours. Alternatively, one can also look for neighbours that are located diagonally to a cell. The default is diag = FALSE.

Value

[list of integer(3)].
List of neighbours. Each list element stands for a combination of predecessing, current and succeeding cell.

Examples

```
cell.ids = c(5, 84, 17, 23)
blocks = c(5, 4, 7)

# (1) Considering diagonal neighbours as well:
findLinearNeighbours(cell.ids = cell.ids, blocks = blocks, diag = TRUE)

# (2) Only consider neighbours which are parellel to the axes:
findLinearNeighbours(cell.ids = cell.ids, blocks = blocks)
```

---

findNearestPrototype *Find Nearest Prototype*

Description

For each cell of the initial design, select the closest observation to its center and use it as a representative for that cell.

Usage

```
findNearestPrototype(feat.object, dist_meth, mink_p, fast_k, ...)
```

Arguments

feat.object	[FeatureObject] A feature object as created by createFeatureObject.
dist_meth	[character(1)] Which distance method should be used for computing the distance between two observations? All methods of dist are possible options with "euclidean" being the default.
mink_p	[integer(1)] Value of p in case dist_meth is "minkowski". The default is 2, i.e. the euclidean distance.
fast_k	[numeric(1)] Percentage of elements that should be considered within the nearest neighbour computation. The default is 0.05.
...	[any] Further arguments, which might be used within the distance computation (dist).

Value

[data.frame].
A data.frame containing one prototype (i.e. a representative observation) per cell. Each prototype consists of its values from the decision space, the corresponding objective value, its own cell ID and the cell ID of the cell, which it represents.

Examples

```
# (1) create the initial sample and feature object:
X = createInitialSample(n.obs = 1000, dim = 2,
  control = list(init_sample.lower = -10, init_sample.upper = 10))
feat.object = createFeatureObject(X = X,
  fun = function(x) sum(x^2), blocks = 10)

# (2) find the nearest prototypes of all cells:
findNearestPrototype(feat.object)
```

`functionInput` *Shiny Server Function for Feature Calculation of Function Input*

Description

`functionInput` is a `shiny` server function which controls all aspects of the `FlaccoFunctionInput` UI Module. Will be called with `callModule`.

Usage

```
functionInput(input, output, session, stringsAsFactors)
```

Arguments

`input`	[`shiny-input`] `shiny` input variable for the specific UI module.
`output`	[`shiny-output object`] `shiny` output variable for the specific UI module.
`session`	[`shiny-session object`] `shiny` session variable for the specific UI module.
`stringsAsFactors`	[`logical(1)`] How should strings be treated internally?

`ggplotFeatureImportance`

Feature Importance Plot

Description

Creates a feature importance plot.

Usage

```
ggplotFeatureImportance(featureList, control = list(), ...)

plotFeatureImportance(featureList, control = list(), ...)
```

Arguments

`featureList`	[`list`] List of vectors of features. One list element is expected to belong to one resampling iteration / fold.
`control`	[`list`] A list, which stores additional configuration parameters: • `featimp.col_{high/medium/low}`: Color of the features, which are used often, sometimes or only a few times. • `featimp.perc_{high/low}`: Percentage of the total number of folds, defining when a features, is used often, sometimes or only a few times. • `featimp.las`: Alignment of axis labels. • `featimp.lab_{feat/resample}`: Axis labels (features and resample iterations). • `featimp.string_angle`: Angle for the features on the x-axis. • `featimp.pch_{active/inactive}`: Plot symbol of the active and inactive points. • `featimp.col_inactive`: Color of the inactive points. • `featimp.col_vertical`: Color of the vertical lines. • `featimp.lab_{title/strip}`: Label used for the title and/or strip label. These parameters are only relevant for `ggplotFeatureImportance`. • `featimp.legend_position`: Location of the legend. This parameter is only relevant for `ggplotFeatureImportance`. • `featimp.flip_axes`: Should the axes be flipped? This parameter is only relevant for `ggplotFeatureImportance`. • `featimp.plot_tiles`: Visualize (non-)selected features with tiles? This parameter is only relevant for `ggplotFeatureImportance`.
`...`	[any] Further arguments, which can be passed to `plot`.

Value

[`plot`].
Feature Importance Plot, indicating which feature was used during which iteration.

Examples

```
## Not run:
# At the beginning, one needs a list of features, e.g. derived during a
# nested feature selection within mlr (see the following 8 steps):
library(mlr)
```

```
library(mlbench)
data(Glass)

# (1) Create a classification task:
classifTask = makeClassifTask(data = Glass, target = "Type")

# (2) Define the model (here, a classification tree):
lrn = makeLearner(cl = "classif.rpart")

# (3) Define the resampling strategy, which is supposed to be used within
# each inner loop of the nested feature selection:
innerResampling = makeResampleDesc("Holdout")

# (4) What kind of feature selection approach should be used? Here, we use a
# sequential backward strategy, i.e. starting from a model with all features,
# in each step the feature decreasing the performance measure the least is
# removed from the model:
ctrl = makeFeatSelControlSequential(method = "sbs")

# (5) Wrap the original model (see (2)) in order to allow feature selection:
wrappedLearner = makeFeatSelWrapper(learner = lrn,
  resampling = innerResampling, control = ctrl)

# (6) Define a resampling strategy for the outer loop. This is necessary in
# order to assess whether the selected features depend on the underlying
# fold:
outerResampling = makeResampleDesc(method = "CV", iters = 10L)

# (7) Perform the feature selection:
featselResult = resample(learner = wrappedLearner, task = classifTask,
  resampling = outerResampling, models = TRUE)

# (8) Extract the features, which were selected during each iteration of the
# outer loop (i.e. during each of the 10 folds of the cross-validation):
featureList = lapply(featselResult$models,
  function(mod) getFeatSelResult(mod)$x)
## End(Not run)

#############################################################################

# Now, one could inspect the features manually:
featureList

# Alternatively, one might use visual means such as the feature
# importance plot. There exist two versions for the feature importance
# plot. One based on the classical R figures
plotFeatureImportance(featureList)

# and one using ggplot
ggplotFeatureImportance(featureList)
```

`listAvailableFeatureSets` *List Available Feature Sets*

Description

Lists all available feature sets w.r.t. certain restrictions.

Usage

```
listAvailableFeatureSets(
  subset,
  allow.cellmapping,
  allow.additional_costs,
  blacklist
)
```

Arguments

`subset`	[`character`] Vector of feature sets, which should be considered. If not defined, all features will be considered.
`allow.cellmapping`	[`logical(1)`] Should (general) cell mapping features be considered as well? The default is `TRUE`.
`allow.additional_costs`	[`logical(1)`] Should feature sets be considered, which require additional function evaluations? The default is `TRUE`.
`blacklist`	[`character`] Vector of feature sets, which should not be considered. The default is `NULL`.

Value

[`character`].
Feature sets, which could be computed - based on the provided input.

Examples

```
sets = listAvailableFeatureSets()
```

measureTime *Measure Runtime of a Feature Computation*

Description

Simple wrapper around `proc.time`.

Usage

```
measureTime(expr, prefix, envir = parent.frame())
```

Arguments

`expr`	[`expression`] Expression of which the time should be measured.
`prefix`	[`character(1)`] Name of the corresponding feature set. Used as a prefix for the runtime.
`envir`	[`environment`] Environment in which expr should be evaluated.

Value

Returns the value(s) of the evaluated `expr` and adds two additional attributes: the number of function evaluations `costs_fun_evals` and the runtime `costs_runtime`, which was required for evaluating the expression.

plotBarrierTree2D *Plot Barrier Tree in 2D*

Description

Creates a 2D image containing the barrier tree of this cell mapping.

Usage

```
plotBarrierTree2D(feat.object, control)
```

Arguments

`feat.object`	[`FeatureObject`] A feature object as created by `createFeatureObject`.
`control`	[`list`] A `list`, which stores additional control arguments. For further information, see details.

Details

Possible `control` arguments are:

- Computation of Cell Mapping:
  - `gcm.approach`: Which approach should be used when computing the representatives of a cell. The default is `"min"`, i.e. the observation with the best (minimum) value within per cell.
  - `gcm.cf_power`: Theoretically, we need to compute the canonical form to the power of infinity. However, we use this value as approximation of infinity. The default is `256`.
- Plot Control:
  - `bt.cm_surface`: Should the underlying surface be based on a cell mapping plot (default is `TRUE`)? Alternatively, the cells would be coloured in shades of grey - according to their objective values.
  - `bt.margin`: Margins of the plot as used by `par("mar")`. The default is `c(5, 5, 4, 4)`.
  - `bt.color_surface`: Color of the surface of the perspective plot. The default is `"lightgrey"`.
  - `bt.color_branches`: Color used for the branches of the barrier tree. Per default there will be one color per level.
  - `bt.pch_root`: Symbol used for plotting the root. The default is `17` (filled triangle).
  - `bt.pch_breakpoint`: Symbol used for plotting a breakpoint. The default is `5` (non-filled diamond).
  - `bt.pch_basin`: Symbol used for plotting the leaf (i.e. a basin) of the barrier tree. The default is `19` (filled circle).
  - `bt.col_root`: Color of the root symbol. The default is `"red"`.
  - `bt.lwd`: Width of the lines used for plotting the branches of a barrier tree. The default is `2`.
  - `bt.label.{x, y}_coord`: Label of the x-/y-coordinate (below / left side of the plot).
  - `bt.label.{x, y}_id`: Label of the x-/y-cell ID (above / right side of the plot).

Value

[plot].
A 2D image, visualizing the barrier tree of this cell mapping.

Examples

```
# create a feature object
X = createInitialSample(n.obs = 900, dim = 2)
f = smoof::makeAckleyFunction(dimensions = 2)
y = apply(X, 1, f)
feat.object = createFeatureObject(X = X, y = y, fun = f, blocks = c(4, 6))

# plot the corresponing barrier tree
plotBarrierTree2D(feat.object)
```

plotBarrierTree3D *Plot Barrier Tree in 3D*

Description

Creates a 3D surface plot containing the barrier tree of this cell mapping.

Usage

```
plotBarrierTree3D(feat.object, control)
```

Arguments

feat.object	[FeatureObject] A feature object as created by createFeatureObject.
control	[list] A list, which stores additional control arguments. For further information, see details.

Details

Possible control arguments are:

- Computation of Cell Mapping:
  - gcm.approach: Which approach should be used when computing the representatives of a cell. The default is "min", i.e. the observation with the best (minimum) value within per cell.
  - gcm.cf_power: Theoretically, we need to compute the canonical form to the power of infinity. However, we use this value as approximation of infinity. The default is 256.
- Plot Control:
  - bt.margin: Margins of the plot as used by par("mar"). The default is c(0.5, 1, 0, 0).
  - bt.color_surface: Color of the surface of the perspective plot. The default is "lightgrey".
  - bt.color_branches: Color used for the branches of the barrier tree. Per default there will be one color per level.
  - bt.persp_border: Color of the lines / borders around each facet of the perspective plot. The default is "grey".
  - bt.persp_shade: A ratio defining the shade of the surface. The default is 0.35.
  - bt.persp_{theta, phi}: Angles (in degree) defining the viewing direction of the perspective plot. theta corresponds to the azimuthal direction (default: 330) and phi to the colatitude (default: 15).
  - bt.persp_{xlab, ylab, zlab}: Labels of the x-, y- and z- axis. The defaults are expression(x[1]), expression(x[2]) and expression(f(x[1], x[2]))
  - bt.persp_ticktype: Should the values of each dimension be shown in detail ("detailed") or just via "simple" arrows in direction of increasement along the axes? The default is "detailed".

- `bt.col_root`: Color of the root symbol. The default is `"red"`.
- `bt.pch_root`: Symbol used for plotting the root. The default is `17` (filled triangle).
- `bt.pch_breakpoint`: Symbol used for plotting a breakpoint. The default is `5` (non-filled diamond).
- `bt.pch_basin`: Symbol used for plotting the leaf (i.e. a basin) of the barrier tree. The default is `19` (filled circle).
- `bt.lwd`: Width of the lines used for plotting the branches of a barrier tree. The default is `2`.

Value

[plot].
A 3D-surface plot, visualizing the barrier tree of this cell mapping.

Examples

```
# create a feature object
X = createInitialSample(n.obs = 900, dim = 2)
f = smoof::makeAckleyFunction(dimensions = 2)
y = apply(X, 1, f)
feat.object = createFeatureObject(X = X, y = y, fun = f, blocks = c(4, 6))

# plot the corresponing barrier tree
plotBarrierTree3D(feat.object)
```

---

plotCellMapping *Plot Cell Mapping*

Description

Visualizes the transitions among the cells in the General Cell Mapping approach.

Usage

```
plotCellMapping(feat.object, control)
```

Arguments

`feat.object`	[FeatureObject] A feature object as created by `createFeatureObject`.
`control`	[list] A `list`, which stores additional control arguments. For further information, see details.

Details

Possible `control` arguments are:

- Computation of GCM Features:
  - `gcm.approach`: Which approach should be used when computing the representatives of a cell. The default is `"min"`, i.e. the observation with the best (minimum) value within per cell.
  - `gcm.cf_power`: Theoretically, we need to compute the canonical form to the power of infinity. However, we use this value as approximation of infinity. The default is `256`.
- Plot Control:
  - `gcm.margin`: The margins of the plot as used by `par("mar")`. The default is `c(5, 5, 4, 4)`.
  - `gcm.color_attractor`: Color of the attractors. The default is `"#333333"`, i.e. dark grey.
  - `gcm.color_uncertain`: Color of the uncertain cells. The default is `"#cccccc"`, i.e. grey.
  - `gcm.color_basin`: Color of the basins of attraction. This has to be a function, which computes the colors, depending on the number of attractors. The default is the color scheme from `ggplot2`.
  - `gcm.plot_arrows`: Should arrows be plotted? The default is `TRUE`.
  - `gcm.arrow.length_{x, y}`: Scaling factor of the arrow length in x- and y-direction. The default is `0.9`, i.e. 90% of the actual length.
  - `gcm.arrowhead.{length, width}`: Scaling factor for the width and length of the arrowhead. Per default (`0.1`) the arrowhead is 10% of the length of the original arrow.
  - `gcm.arrowhead.type`: Type of the arrowhead. Possible options are `"simple"`, `"curved"`, `"triangle"` (default), `"circle"`, `"ellipse"` and `"T"`.
  - `gcm.color_grid`: Color of the grid lines. The default is `"#333333"`, i.e. dark grey.
  - `gcm.label.{x, y}_coord`: Label of the x-/y-coordinate (below / left side of the plot).
  - `gcm.label.{x, y}_id`: Label of the x-/y-cell ID (above / right side of the plot).
  - `gcm.plot_{coord, id}_labels`: Should the coordinate (bottom and left) / ID (top and right) labels be plotted? The default is `TRUE`.

Value

[plot].

References

- Kerschke, P., Preuss, M., Hernandez, C., Schuetze, O., Sun, J.-Q., Grimme, C., Rudolph, G., Bischl, B., and Trautmann, H. (2014): "Cell Mapping Techniques for Exploratory Landscape Analysis", in: EVOLVE – A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V, pp. 115-131 (http://dx.doi.org/10.1007/978-3-319-07494-8_9).

Examples

```
# (1) Define a function:
library(smoof)
```

```
f = makeHosakiFunction()

# (2) Create a feature object:
X = cbind(
  x1 = runif(n = 100, min = -32, max = 32),
  x2 = runif(n = 100, min = 0, max = 10)
)
y = apply(X, 1, f)
feat.object = createFeatureObject(X = X, y = y, blocks = c(4, 6))

# (3) Plot the cell mapping:
plotCellMapping(feat.object)
```

`plotInformationContent` *Plot Information Content*

Description

Creates a plot of the Information Content Features.

Usage

```
plotInformationContent(feat.object, control)
```

Arguments

`feat.object`	[FeatureObject] A feature object as created by `createFeatureObject`.
`control`	[list] A `list`, which stores additional control arguments. For further information, see details.

Details

Possible `control` arguments are:

- Computation of Information Content Features:
  - `ic.epsilon`: Epsilon values as described in section V.A of Munoz et al. (2015). The default is `c(0, 10^(seq(-5, 15, length.out = 1000))`.
  - `ic.sorting`: Sorting strategy, which is used to define the tour through the landscape. Possible values are `"nn"` (= default) and `"random"`.
  - `ic.sample.generate`: Should the initial design be created using a LHS? The default is `FALSE`, i.e. the initial design from the feature object will be used.
  - `ic.sample.dimensions`: Dimensions of the initial sample, if created using a LHS. The default is `feat.object$dimension`.
  - `ic.sample.size`: Size of the initial sample, if created using a LHS. The default is `100 * feat.object$dimension`.

  - `ic.sample.lower`: Lower bounds of the initial sample, if created with a LHS. The default is `100 * feat.object$lower`.
  - `ic.sample.upper`: Upper bounds of the initial sample, if created with a LHS. The default is `100 * feat.object$upper`.
  - `ic.show_warnings`: Should warnings be shown, when possible duplicates are removed? The default is `FALSE`.
  - `ic.seed`: Possible seed, which can be used for making your experiments reproducable. Per default, a random number will be drawn as seed.
  - `ic.nn.start`: Which observation should be used as starting value, when exploring the landscape with the nearest neighbour approach. The default is a randomly chosen integer value.
  - `ic.nn.neighborhood`: In order to provide a fast computation of the features, we use `RANN::nn2` for computing the nearest neighbors of an observation. Per default, we consider the `20L` closest neighbors for finding the nearest not-yet-visited observation. If all of those neighbors have been visited already, we compute the distances to the remaining points separately.
  - `ic.settling_sensitivity`: Threshold, which should be used for computing the "settling sensitivity". The default is `0.05` (as used in the corresponding paper).
  - `ic.info_sensitivity`: Portion of partial information sensitivity. The default is `0.5` (as used in the paper).

- Plot Control:
  - `ic.plot.{xlim, ylim, las, xlab, ylab}`: Settings of the plot in general, cf. `plot.default`.
  - `ic.plot.{xlab_line, ylab_line}`: Position of `xlab` and `ylab`.
  - `ic.plot.ic.{lty, pch, cex, pch_col}`: Type, width and colour of the line visualizing the "Information Content" $H(\epsilon)$.
  - `ic.plot.max_ic.{lty, pch, lwd, cex, line_col, pch_col}`: Type, size and colour of the line and point referring to the "Maximum Information Content" $H[max]$.
  - `ic.plot.settl_sens.{pch, cex, col}`: Type, size and colour of the point referring to the "Settling Sensitivity" $\epsilon[s]$.
  - `ic.plot.partial_ic`: Should the information of the partial information content be plotted as well? The default is `TRUE`.
  - `ic.plot.partial_ic.{lty, pch, lwd, cex, line_col, pch_col}`: Type, size and colour of the line and point referring to the "Initial Partial Information" $M[0]$ and the "Partial Information Content" $M(\epsilon)$.
  - `ic.plot.half_partial.{pch, cex, pch_col}`: Type, size and colour of the point referring to the "Relative Partial Information Sensitivity" $\epsilon[ratio]$.
  - `ic.plot.half_partial.{lty, line_col, lwd}_{h, v}`: Type, colour and width of the horizontal and vertical lines referring to the "Relative Partial Information Sensitivity" $\epsilon[ratio]$.
  - `ic.plot.half_partial.text_{cex, col}`: Size and colour of the text at the horizontal line of the "Relative Partial Information Sensitivity" $\epsilon[ratio]$.
  - `ic.plot.legend_{descr, points, lines, location}`: Description, points, lines and location of the legend.

Value

[plot].
A plot visualizing the Information Content Features.

References

- Munoz, M. A., Kirley, M., and Halgamuge, S. K. (2015): "Exploratory Landscape Analysis of Continuous Space Optimization Problems Using Information Content", in: IEEE Transactions on Evolutionary Computation (19:1), pp. 74-87 (http://dx.doi.org/10.1109/TEVC.2014.2302006).

Examples

```
# (1) create a feature object:
X = t(replicate(n = 2000, expr = runif(n = 5, min = -10, max = 10)))
feat.object = createFeatureObject(X = X, fun = function(x) sum(x^2))

# (2) plot its information content features:
plotInformationContent(feat.object)
```

---

runFlaccoGUI *Run the flacco-GUI based on Shiny*

Description

runFlaccoGUI starts a shiny application, which allows the user to compute the flacco features and also visualize the underlying functions.

Usage

```
runFlaccoGUI()
```

Details

A shiny application is a web-app which can be accessed through a browser.

References

- Hanster, C., and Kerschke, P. (2017): "flaccogui: Exploratory Landscape Analysis for Everyone", in: Proceedings of the 19th Annual Conference on Genetic an Evolutionary Computation (GECCO) Companion, pp. 1215-1222, ACM. (http://dl.acm.org/citation.cfm?doid=3067695.3082477).
- Kerschke, P., and Trautmann, H. (2019): "Comprehensive Feature-Based Landscape Analysis of Continuous and Constrained Optimization Problems Using the R-package flacco", in: Applications in Statistical Computing – From Music Data Analysis to Industrial Quality Improvement, pp. 93-123, Springer. (https://link.springer.com/chapter/10.1007/978-3-030-25147-5_7).

SmoofImport *Shiny Server Function for BBOB Import Page Module*

Description

SmoofImport is a shiny server function which will control all aspects of the SmoofImportPage-UI Module. It will be called with callModule.

Usage

```
SmoofImport(input, output, session, stringsAsFactors)
```

Arguments

input	[shiny-input] shiny input variable for the specific UI module.
output	[shiny-output object] shiny output variable for the specific UI module.
session	[shiny-session object] shiny session variable for the specific UI module.
stringsAsFactors	[logical(1)] How should strings be treated internally?

SmoofImportPage *Shiny UI-Module for Batch Import of Smoof Functions*

Description

SmoofImportPage is a shiny UI-component which can be added to your shiny app so that you get a batch import for a specific shiny function but different parameters.

Usage

```
SmoofImportPage(id)
```

Arguments

id	[character(1)] Character representing the namespace of the shiny component.

Details

It will load a CSV-file with parameters for the smoof function and calculate the selected features for the specific function.

Index

∗ **data**
featureList, 18

BBOBImport, 2
BBOBImportPage, *2*, 3

calculateFeatures *(*calculateFeatureSet*)*, 3
calculateFeatureSet, 3
callModule, *2*, *20*, *22*, *25*, *37*
character, *3*, *19–21*, *23, 24*, *28*, *37*
computeGridCenters, 15
convertInitDesignToGrid, 16
cor, *8*
createFeatureObject, *4*, *21, 22*, *24*, *29*, *31, 32*, *34*
createFeatureObject *(*FeatureObject*)*, 18
createInitialSample, 17
cutree, *7*

data.frame, *16*, *19*, *24*
density, *4*, *6*
dist, *5*, *8*, *24*

featureList, 18
FeatureObject, *4*, *18*, 18, *19*, *21, 22*, *24*, *29*, *31, 32*, *34*
featureObject_sidebar, 20
FeatureSetCalculation, 20
FeatureSetCalculationComponent, 21
FeatureSetVisualization, 22
FeatureSetVisualizationComponent, 22
findLinearNeighbours, 23
findNearestPrototype, 24
function, *19*
functionInput, 25

ggplotFeatureImportance, 25
Glass, *18*
grad, *6*

hclust, *7*
hessian, *6*

integer, *15, 16*, *19*, *23, 24*

kurtosis, *6*

list, *23*
listAvailableFeatureSets, *4*, 28
logical, *3*, *19*, *21–23*, *25*, *28*, *37*

matrix, *17*, *19*
measureTime, 29

numeric, *15, 16*, *19*, *24*

optim, *4*

plot.default, *35*
plotBarrierTree2D, 29
plotBarrierTree3D, 31
plotCellMapping, 32
plotFeatureImportance *(*ggplotFeatureImportance*)*, 25
plotInformationContent, 34

ResampleDesc, *6*
runFlaccoGUI, 36

skewness, *6*
SmoofImport, 37
SmoofImportPage, 37