Chapter 7
Comprehensive Feature-Based Landscape Analysis of Continuous and Constrained Optimization Problems Using the R-Package Flacco

Pascal Kerschke and Heike Trautmann

Abstract Choosing the best-performing optimizer(s) out of a portfolio of optimization algorithms is usually a difficult and complex task. It gets even worse, if the underlying functions are unknown, i.e., so-called *black-box problems*, and function evaluations are considered to be expensive. In case of continuous single-objective optimization problems, *exploratory landscape analysis (ELA)*, a sophisticated and effective approach for characterizing the landscapes of such problems by means of numerical values before actually performing the optimization task itself, is advantageous. Unfortunately, until now it has been quite complicated to compute multiple ELA features simultaneously, as the corresponding code has been—if at all—spread across multiple platforms or at least across several packages within these platforms. This article presents a broad summary of existing ELA approaches and introduces `flacco`, an `R`-package for **f**eature-based **l**andscape **a**nalysis of **c**ontinuous and **c**onstrained **o**ptimization problems. Although its functions neither solve the optimization problem itself nor the related *algorithm selection problem (ASP)*, it offers easy access to an essential ingredient of the ASP by providing a wide collection of ELA features on a single platform—even within a single package. In addition, `flacco` provides multiple visualization techniques, which enhance the understanding of some of these numerical features, and thereby make certain landscape properties more comprehensible. On top of that, we will introduce the package's built-in, as well as web-hosted and hence platform-independent, graphical user interface (GUI). It facilitates the usage of the package—especially for people who are not familiar with `R`—and thus makes `flacco` a very convenient toolbox when working towards algorithm selection of continuous single-objective optimization problems.

P. Kerschke (✉) · H. Trautmann
Information Systems and Statistics, University of Münster, Leonardo-Campus 3, 48149 Münster, Germany
e-mail: kerschke@uni-muenster.de

H. Trautmann
e-mail: trautmann@uni-muenster.de

N. Bauer et al. (eds.), *Applications in Statistical Computing*,
Studies in Classification, Data Analysis, and Knowledge Organization,
https://doi.org/10.1007/978-3-030-25147-5_7

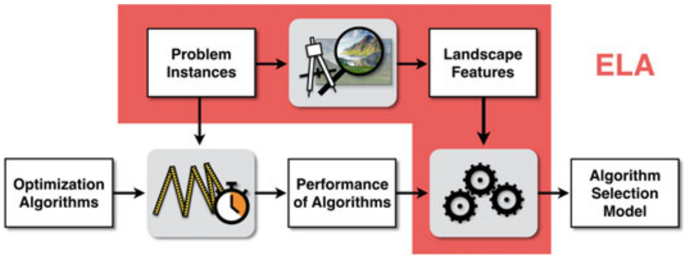


Fig. 7.1 Scheme for using ELA in the context of an algorithm selection problem

7.1 Introduction

Already a long time ago, Rice introduced the *algorithm selection problem (ASP)* (Rice 1976), which aims at finding the best algorithm A out of a set of algorithms $\mathscr{A}$ for a specific instance[1] $I \in \mathscr{I}$. Thus, one can say that the algorithm selection model m tries to find the best-performing algorithm A for a given set of problem instances $\mathscr{I}$.

As the performance of an optimization algorithm strongly depends on the structure of the underlying instance, it is crucial to have some a priori knowledge of the problem's landscape in order to pick a well-performing optimizer. One promising approach for such heads-up information are landscape features, which characterize certain landscape properties by means of numerical values. Figure 7.1 schematically shows how *exploratory landscape analysis (ELA)* can be integrated into the model-finding process of an ASP. In the context of continuous single-objective optimization, relevant problem characteristics cover information such as the degree of *multimodality*, its *separability* or its *variable scaling* (Mersmann et al. 2011). However, the majority of these so-called "high-level properties" have a few drawbacks: they (1) are categorical, (2) require expert knowledge (i.e., a decision maker), (3) might miss relevant information, and (4) require knowledge of the entire (often unknown) problem. Therefore, Bischl et al. (2012) introduced so-called "low-level properties", i.e., the landscape features, which characterize a problem instance by means of—not necessarily intuitively understandable—numerical values, based on a sample of observations from the underlying problem. Figure 7.2 shows which of their automatically computable low-level properties (shown as white ellipses) could potentially describe the expert-designed high-level properties (grey boxes).

In recent years, researchers all over the world have developed further (low-level) landscape features and implemented them in many different programming languages—mainly in `MATLAB` (2013), `Python` (VanRossum and The Python Development Team 2015), or `R` (R Core Team 2018). Most of these features are very cheap to compute as they only require an initial design, i.e., a small amount of

[1]In this context, an *instance* is the equivalent to an optimization problem, i.e., it maps the elements of the decision space $\mathscr{X}$ to the objective space $\mathscr{Y}$.

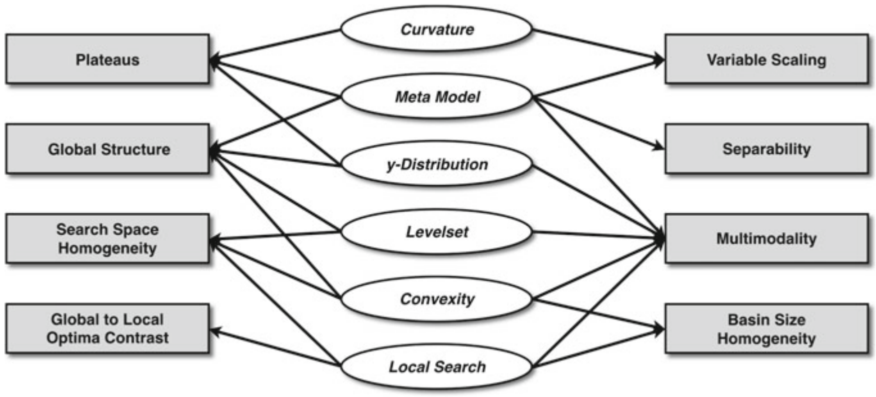


Fig. 7.2 Relationship between expert-based high-level properties (gray rectangles in the outer columns) and automatically computable low-level properties (white ellipses in the middle)

(often randomly chosen) exemplary observations, which are used as representatives of the original problem instance.

For instance, Jones and Forrest (1995) characterized problem landscapes using the *correlation* between the distances from a set of points to the global optimum and their respective fitness values. Later, Lunacek and Whitley (2006) introduced *dispersion* features, which compare pairwise distances of all points in the initial design with the pairwise distances of the best points in the initial design (according to the corresponding objective). Malan and Engelbrecht (2009) designed features that quantify the *ruggedness* of a continuous landscape and Müller and Sbalzarini (2011) characterized landscapes by performing *fitness-distance analyses*. In other works, features were constructed using the problem definition, hill climbing characteristics and a set of random points (Tinus et al. 2013), or based on a tour across the problem's landscape, using the change in the objective values of neighboring points to measure the landscape's *information content* (Muñoz et al. 2012, 2015a).

The fact that feature-based landscape analysis also exists in other domains—e.g., in discrete optimization (Daolio et al. 2016; Jones 1995; Ochoa et al. 2014) including its subdomain, the *traveling salesperson problem* (Mersmann et al. 2013; Hutter et al. 2014; Pihera and Musliu 2014)—also led to attempts to discretize the continuous problems and afterward use a so-called *cell mapping* approach (Kerschke et al. 2014), or compute *barrier trees* on the discretized landscapes (Flamm et al. 2002).

In recent years, Morgan and Gallagher (2015) analyzed optimization problems by means of *length scale* features, Kerschke et al. (2015, 2016) distinguished between so-called "funnel" and "non-funnel" structures with the help of *nearest better clustering* features, Malan et al. (2015) used *violation landscapes* to characterize constrained continuous optimization problems, and (Shirakawa and Nagao 2016) introduced the *bag of local landscape features*.

Looking at the list from above, one can see that there exists a plethora of approaches for characterizing the fitness landscapes of continuous optimization problems, which makes it difficult to keep track of all the new approaches. For this purpose, we would like to refer the interested reader to the survey paper by Muñoz et al. (2015b), which provides a nice introduction and overview on various methods and challenges of this whole research field, i.e., the combination of feature-based landscape analysis and algorithm selection for continuous black-box optimization problems.

Given the studies from above, each of them provided new feature sets, describing certain aspects or properties of a problem. However, due to the fact that one would have to run the experiments across several platforms, usually only a few feature sets were combined—if at all—within a single study. As a consequence, the full potential of feature-based landscape analysis could not be exploited. This issue has now been addressed with the implementation of the `R`-package `flacco` (Kerschke 2017), which is introduced within this paper.

While the general idea of `flacco` (Kerschke and Trautmann 2016), as well as its associated graphical user interface (Hanster and Kerschke 2017) were already published individually in earlier works, this paper combines them for the first time and—more importantly—clearly enhances them by providing a detailed description of the implemented landscape features.

The following section provides a very detailed overview of the integrated landscape features and the additional functionalities of `flacco`. Section 7.3 demonstrates the package usage based on a well-known optimization problem and Sect. 7.4 introduces the built-in visualization techniques. In Sect. 7.5, the functionalities of the package's graphical user interface are presented and Sect. 7.6 concludes this work.

7.2 Integrated ELA Features

The here presented `R`-package provides a unified interface to a collection of the majority of feature sets from above within a single package[2] and consequently simplifies their accessibility. In its default settings, `flacco` computes more than 300 different numerical landscape features, distributed across 17 so-called *feature sets* (further details on these sets are given on p. xx). Additionally, the total number of performed function evaluations, as well as the required runtime for the computation of a feature set, are automatically tracked per feature set.

Within `flacco`, basically all feature computations and visualizations are based on a *feature object*, which stores all the relevant information of the problem instance. It contains the initial design, i.e., a data frame of all the (exemplary) observations from the decision space along with their corresponding objective values and—if provided—the number of blocks per input dimension (e.g., required for the *cell mapping* approach, cf. Kerschke et al. 2014), as well as the exact (mathematical)

[2]The authors intend to extend `flacco` by any feature set that has not yet been integrated into it.

function definition, which is needed for those feature sets, which perform additional function evaluations, e.g., the *local search* features (which were already mentioned in Fig. 7.2).

Such a feature object can be created in different ways as shown in Fig. 7.3. First, one has to provide some input data by either passing (a) a matrix or data frame of the (sampled) observations from the decision space (denoted as `X`), as well as a vector with the corresponding objective values (`y`), or (b) a data frame of the initial design (`init.design`), which basically combines the two ingredients from (a), i.e., the decision space samples and the corresponding objective values, within a single data set. Then, the function `createFeatureObject` takes the provided input data and transforms it into a specific `R` object—the feature object. During its creation, this object can (optionally) be enhanced/specified by further details such as the exact function of the problem (`fun`), the number of blocks per dimension (`blocks`), or the upper (`upper`) and lower (`lower`) bounds of the problem's decision space. If the latter are not provided, the initial design's minimum and maximum (per input variable) will be used as boundaries.

The package's function `createInitialSample` allows to create a random sample of observations (denoted as `X` within Fig. 7.3) within the box-constraints $[0, 1]^d$ (with d being the number of input dimensions). This sample-generator also allows to configure the boundaries or use an *improved latin hypercube sample* (Beachkofski and Grandhi 2002; McKay et al. 2000) instead of a sample that is based on a random uniform distribution.

As one of the main purposes of *exploratory landscape analysis* is the description of landscapes when given a very restricted budget (of function evaluations), it is highly

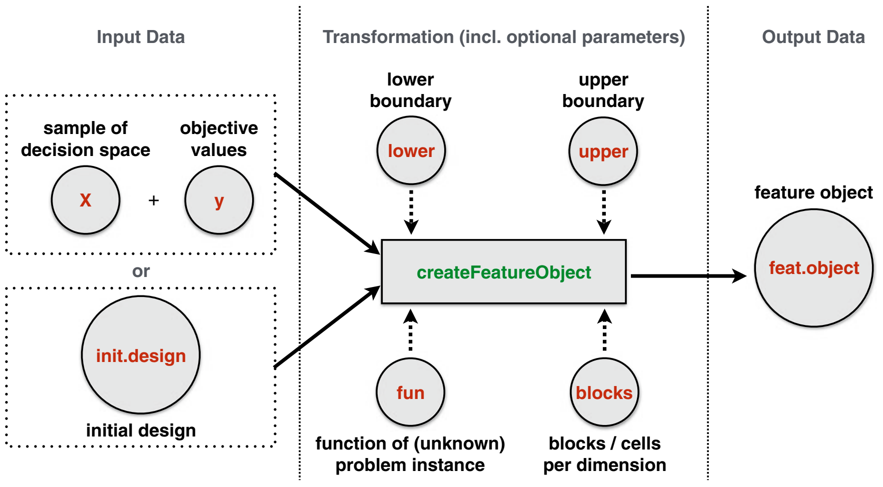


Fig. 7.3 Scheme for creating a feature object: (1) provide the input data, (2) call `createFeatureObject` and pass additional arguments if applicable, (3) receive the feature object, i.e., a specific `R`-object, which is the basis for all further computations and visualizations

recommended to keep the sample sizes small. Within recent work, Kerschke et al. (2016) have shown that initial designs consisting of $50 \cdot d$ observations—i.e., sampled points in the decision space—can be completely sufficient for describing certain high-level properties by means of numerical features. Also, one should note that the majority of research within the field of continuous single-objective optimization deals with at most 40-dimensional problems. Thus, unless one possesses a high budget of function evaluations, as well as the computational resources, it is not useful to compute features based on initial designs with more than 10,000 observations or more than 40 features. And even in case one is in possession of such resources, one should consider whether it is necessary to perform feature-based algorithm selection; depending on the problems, algorithms, and resources, it might as well be possible to simply run all optimization algorithms on all problem instances.

Given the aforementioned feature object, one has laid the basis for calculating specific feature sets (using the function `calculateFeatureSet`), computing all available feature sets at once (`calculateFeatures`) or visualizing certain characteristics (described in more detail in Sect. 7.4).

In order to get an overview of all currently implemented feature sets, one can call the function `listAvailableFeatureSets`. This function, as well as `calculateFeatures`, i.e., the function for calculating all feature sets at once, allows to turn off specific feature sets, such as those, which require additional function evaluations or follow the *cell mapping* approach. Turning off the *cell mapping* features could, for instance, be useful in case of higher dimensional problems due to the *curse of dimensionality*[3] (Friedman 1997).

In general, the current version of `flacco` consists of the feature sets that are described below. Note that the final two "features" per feature set always provide its costs, i.e., the additionally performed function evaluations and running time (in seconds) for calculating that specific set.

Classical ELA Features (83 features across 6 feature sets):
The first six groups of features are the "original" ELA features as defined by Mersmann et al. (2011).

Six features measure the *convexity* by taking multiple times (default: 1,000) two points of the initial design and comparing the convex combination of their objective values with the objective value of their convex combinations. Obviously, each convex combination requires an additional function evaluation. The set of these differences—i.e., the distance between the objective value of the convex combination and the convex combination of the objective values—is then used to estimate the probabilities of convexity and linearity. In addition, the arithmetic mean of these differences, as well as the arithmetic mean of the absolute differences are used as features as well.

The 26 *curvature* features measure information on the estimated gradient and Hessian of (a subset of) points from the initial design. More precisely, the lengths' of the gradients, the ratios between the biggest and smallest (absolute) gradient directions and the ratios of the biggest and smallest eigenvalues of the Hessian matrices

[3]In case of a 10-dimensional problem in which each input variable is discretized by three blocks, one already needs $3^{10} = 59{,}049$ observations to have one observation per cell—on average.

are aggregated using the minimum, first quartile (= 25%-quantile), arithmetic mean, median, third quartile (= 75%-quantile), maximum, standard deviation and number of missing values.[4] Note that estimating the derivatives also requires additional function evaluations (default: $100 \times d$ with search space dimensionality d).

The five *y-distribution* features compute the kurtosis, skewness and number of peaks based on a kernel-density estimation of the initial design's objective values.

For the next group, the initial design is divided into two groups (based on a configurable threshold for the objective values). Afterward the performances of various classification techniques, which are applied to that binary classification task, are used for computing 20 *levelset* features. Based on the classifiers and thresholds,[5] `flacco` uses the `R`-package `mlr` (Bischl et al. 2016) to compute the mean misclassification errors (per combination of classifier and threshold). Per default, the error rates are based on a tenfold cross-validation, but this can be changed to any other resampling-strategy implemented in `mlr`. In addition to the "pure" misclassification errors, `flacco` computes the ratio of misclassification errors per threshold and pair of classifiers.

The 16 *local search* features extract information from several local search runs (each of them starting in one of the points from the initial design). Per default, we run $50 \times d$ local searches with the quasi-Newton method `L-BFGS-B` (Byrd et al. 1995) (as implemented in `R`'s `optim`-function) to find a local optimum. The found optima are then clustered (per default via single-linkage hierarchical clustering, see, e.g., Jobson 2012) in order to decide whether the found optima belong to the same basins. The resulting total number of found optima (= clusters) is the first feature of this group, whereas the ratio of this number of optima and the conducted local searches form the second one. The next two features measure the ratio of the best and average objective values of the found clusters. As the clusters likely combine multiple local optima, the number of optima per cluster can be used as a representation for the basin sizes. These basin sizes are then averaged for the best, all non-best and the worst cluster(s). Finally, the number of required function evaluations across the different local search runs is aggregated using the minimum, first quartile (= 25%-quantile), arithmetic mean, median, third quartile (= 75%-quantile), maximum and standard deviation.

The remaining 11 *meta model* features extract information from linear and quadratic models (with and without interactions) that were fitted to the initial design. More precisely, these features compute the model fit ($= R^2_{adj}$) for each of the four aforementioned models, the intercept of the linear model without interactions, the condition (i.e., the ratio of largest and smallest absolute coefficient) of the quadratic model without interactions and the minimum, maximum as well as ratio of maximum

[4]If a point is a local optimum the gradient is zero for all dimensions of a sample point, then the ratio of biggest and smallest gradient obviously cannot be computed and therefore results in a missing value (= `NA`).

[5]The default classifiers are linear (LDA), quadratic (QDA) and mixture discriminant analysis (MDA) and the default threshold for dividing the data set into two groups are the 10%-, 25%- and 50%-quantile of the objective values.

and minimum of all (absolute) coefficients—except for the intercept—of the linear model without interactions.

Cell Mapping Features (20/3):
This approach discretizes the continuous decision space using a predefined number of blocks (= cells) per input dimension. Then, the interaction of the cells, as well as the observations that are located within a cell, are used to compute the corresponding landscape features (Kerschke et al. 2014).

As shown in Fig. 7.4, the ten *angle* features extract information based on the location of the best and worst (available) observation of a cell w.r.t. the corresponding cell center. The distance from cell center to the best and worst point within the cell, as well as the angle between the three points are computed per cell and afterward aggregated across all cells using the arithmetic mean and standard deviation. The remaining features compute the difference between the best and worst objective value per cell, normalize these differences by the biggest span in objective values within the entire initial design and afterward aggregate these distances across all cells (using the arithmetic mean and the standard deviation).

The four *gradient homogeneity* features aggregate the cell-wise information on the gradients between each point of a cell and its corresponding nearest neighbor. Figure 7.5 illustrates the idea of this feature set. That is, for each point within a cell, we compute the gradient towards its nearest neighbor, normalize it, point it towards the better one of the two neighboring points and afterwards sum up all the normalized gradients (per cell). Then, the lengths of the resulting vectors are aggregated across all cells using the arithmetic mean and standard deviation.

The remaining six cell mapping features aggregate the (estimated) *convexity* based on representative observations of (all combinations of) three either horizontally, vertically or diagonally successive cells. Each cell is represented by the sample obser-

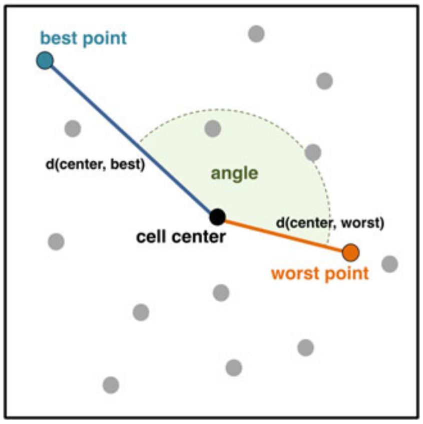


Fig. 7.4 Overview of the “ingredients” computing the *cell mapping angle features*: the location of the best and worst points within a cell

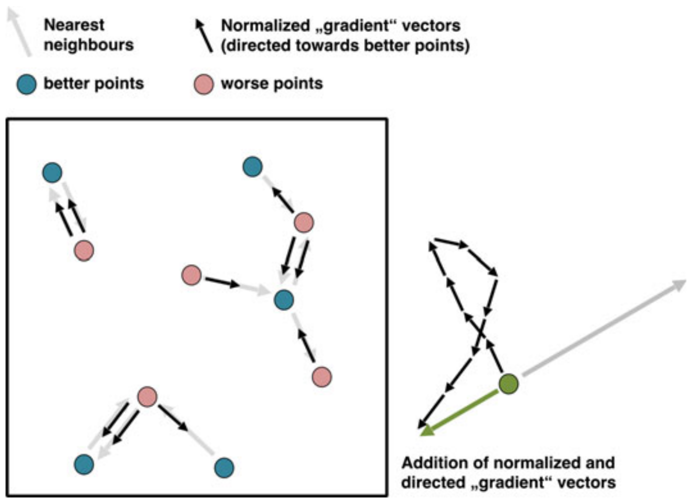

Nearest neighbours
Normalized „gradient“ vectors (directed towards better points)
better points
worse points
Addition of normalized and directed „gradient“ vectors

Fig. 7.5 General idea for computing the *cell mapping gradient homogeneity features*: (1) find the nearest neighbors, (2) compute and normalize the gradient vectors, (3) point them towards the better of the two points and (4) sum them up

vation that is located closest to the corresponding cell center. We then compare—for each combination of three neighboring cells—their objective values: if the objective value of the middle cell is below/above the mean of the objective values of the outer two cells, we have an indication for (soft) convexity/concavity. If the objective value of the middle cell even is the lowest/biggest value of the three neighboring cells, we have an indication for (hard) convexity/concavity. Averaging these values across all combinations of three neighboring cells, results in the estimated "probabilities" for (soft/hard) convexity or concavity.

Generalized Cell Mapping Features (75/1):
Analogously to the previous group of features, these features are also based on the block-wise discretized decision space. Here, each cell will be represented by exactly one of its observations—either its best or average value or the one that is located closest to the cell center—and each of the cells is then considered to be an absorbing Markov chain. That is, for each cell the transition probability for moving from one cell to one of its neighboring cells is computed. Based on the resulting transition probabilities, the cells are categorized into attractor, transient, periodic and uncertain cells. Figure 7.6 shows two exemplary cell mapping plots—each of them is based on the same feature object, but follows a different representation approach—which color the cells according to their category: attractor cells are depicted by black boxes, gray cells indicate uncertain cells, i.e., cells that are attracted by multiple attractors, and the remaining cells represent the certain cells, which form the basins of attractions. Furthermore, all non-attractor cells possess arrows that point toward their attractors and their length's represent the attraction probabilities. The different cell types, as

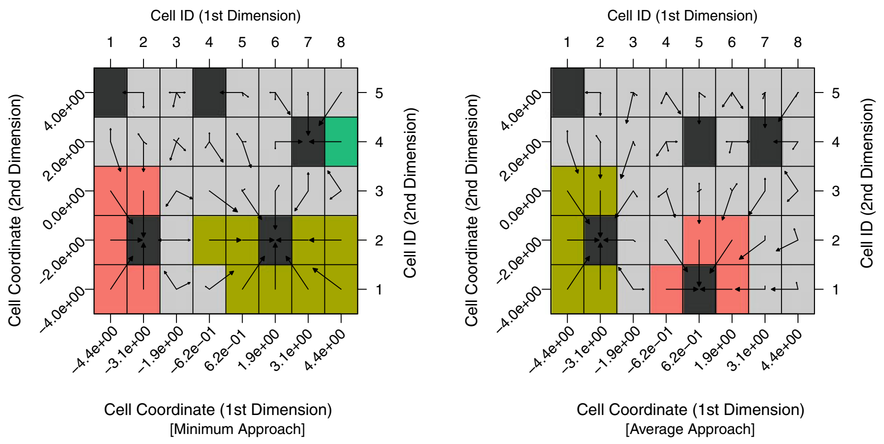


Fig. 7.6 Cell mappings of a two-dimensional version of *Gallagher's Gaussian 101-me Peaks* (Hansen et al. 2009) function based on a minimum (left) and average (right) approach. The black cells are the attractor cells, i.e., potential local optima. Each of those cells might come with a basin of attraction, i.e., the colored cells which point towards the attractor. All uncertain cells, i.e., cells that are attracted by multiple attractors, are shown in grey. The arrows show in the direction(s) of their attracting cell(s)

well as the accompanying probabilities are the foundation for the 25 *generalized cell mapping* (GCM) features (per approach):

1. total number of attractor cells and ratio of cells that are periodic (usually the attractor cells), transient (= nonperiodic cells) or uncertain cells,
2. aggregation (minimum, arithmetic mean, median, maximum and standard deviation) of the probabilities for reaching the different basins of attractions,
3. aggregation of basin sizes when the basins are only formed by the "certain" cells,
4. aggregation of basin sizes when the "uncertain" cells also count toward the basins (a cell which points towards multiple attractors contributes to each of them),
5. number and probability of finding the attractor cell with the best objective value.

Further details on the GCM-approach are given in Kerschke et al. (2014).

Barrier Tree Features (93/1):
Building on top of the transition probability representation and the three approaches from the GCM features, a so-called *barrier tree* (Flamm et al. 2002), as shown in Fig. 7.7, is constructed (per approach). The local optima of the problem landscape—i.e., the valleys in case of minimization problems—are represented by the leaves of the tree (indicated by filled circles within Fig. 7.7) and the branching nodes (depicted as non-filled diamonds) represent the ridges of the landscape, i.e., the locations where (at least) two neighboring valleys are connected. Based on these trees, the following 31 features can be computed for each of the three cell-representing approaches:

1. number of leaves (= filled circles) and levels (= branches of different colors), as well as tree depth, i.e., distance from root node (red triangle) to the lowest leaf,

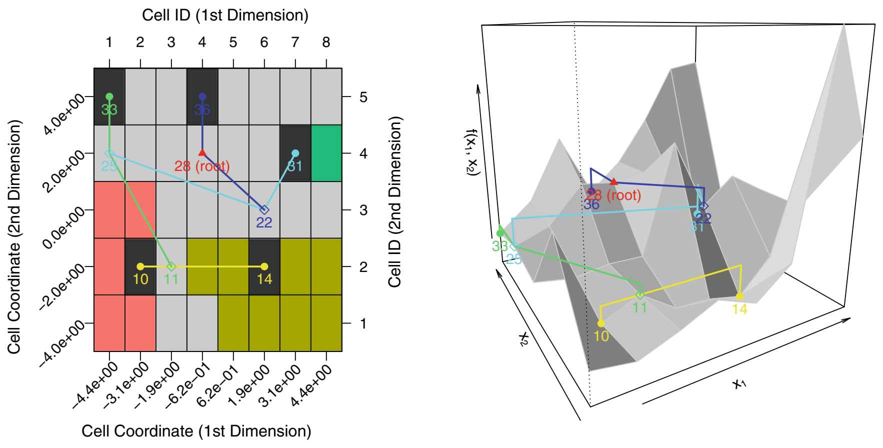


Fig. 7.7 Visualizations of a barrier tree in 2D (left) and 3D (right). Both versions of the tree belong to the same problem—a two-dimensional version of *Gallagher's Gaussian 101-me Peaks* function based on the minimum approach. The root of the tree is highlighted by a red triangle and the leaves of the tree, which usually lie in an attractor cell, are marked with filled circles. The saddle points, i.e., the points where two or more basins of attraction come together, are indicated by non-filled circles, as well as one incoming and two outgoing branches

2. ratio of the depth of the tree and the number of levels,
3. ratio of the number of levels and the number of non-root nodes,
4. aggregation (minimum, arithmetic mean, median, maximum and standard deviation) of the height differences between a node and its predecessor,
5. aggregation of the average height difference per level,
6. ratio of biggest and smallest basin of attraction (based on the number of cells counting towards the basin) for three different "definitions" of a basin: (a) based on the "certain" cells, (b) based on the "uncertain" cells or (c) based on the "certain" cells, plus the "uncertain" cells for which the probability towards the respective attractor is the highest,
7. proportion of cells that belong to the basin, which contains the attractor with the best objective value, and cells from the other basins—aggregated across the different basins,
8. widest range of a basin.

Nearest Better Clustering Features (7/1):
These features extract information based on the comparison of the sets of distances from (a) all observations towards their nearest neighbors and (b) their *nearest better neighbors*[6] (Kerschke et al. 2015). More precisely, these features measure the ratios of the standard deviations and the arithmetic means between the two sets, the correlation between the distances of the nearest neighbors and nearest better neighbors, the

[6]Here, the "nearest better neighbor" is the observation, which is the nearest neighbor among the set of all observations with a better objective value.

coefficient of variation of the distance ratios and the correlation between the fitness value, and the count of observations to whom the current observation is the nearest better neighbor, i.e., the so-called “indegree”.

Dispersion Features (18/1):
The *dispersion features* by Lunacek and Whitley (2006) compare the dispersion among observations within the initial design and among a subset of these points. The subsets are created based on predefined thresholds, whose default values are the 2%-, 5%-, 10%- and 25%-quantile of the objective values. For each of these threshold values, we compute the arithmetic mean and median of all distances among the points of the subset, and then compare it—using the difference and ratio—to the mean or median, respectively, of the distances among all points from the initial design.

Information Content Features (7/1):
The *Information Content of Fitness Sequences (ICoFiS)* approach (Muñoz et al. 2015a) quantifies the so-called *information content* of a continuous landscape, i.e., smoothness, ruggedness, or neutrality. While similar methods already exist for the information content of discrete landscapes, this approach provides an adaptation to continuous landscapes that for instance accounts for variable step sizes in random walk sampling. This approach is based on a symbol sequence $\Phi = \{\phi_1, \dots, \phi_{n-1}\}$, with

$$\phi_i := \begin{cases} \bar{1} & \text{, if} \quad \frac{y_{i+1}-y_i}{||\mathbf{x}_{i+1}-\mathbf{x}_i||} < -\varepsilon \\ 0 & \text{, if} \quad \left|\frac{y_{i+1}-y_i}{||\mathbf{x}_{i+1}-\mathbf{x}_i||}\right| \leq \varepsilon \\ 1 & \text{, if} \quad \frac{y_{i+1}-y_i}{||\mathbf{x}_{i+1}-\mathbf{x}_i||} > \varepsilon \end{cases} .$$

This sequence is derived from the objective values $y_1, \dots, y_n$ belonging to the n points $\mathbf{x}_1, \dots, \mathbf{x}_n$ of a random walk across (the initial design of) the landscape and depends on the *information sensitivity* parameter $\varepsilon > 0$.

This symbol sequence Φ is aggregated by the *information content* $H(\varepsilon) := \sum_{i \neq j} p_{ij} \cdot \log_6 p_{ij}$, where p_{ij} is the probability of having the “block” $\phi_i \phi_j$, with $\phi_i, \phi_j \in \{\bar{1}, 0, 1\}$, within the sequence. Note that the base of the logarithm was set to six as this equals the number of possible blocks $\phi_i \phi_j$ for which $\phi_i \neq \phi_j$, i.e., $\phi_i \phi_j \in \{\bar{1}0, 0\bar{1}, \bar{1}1, 1\bar{1}, 10, 01\}$ (Muñoz et al. 2015a).

Another aggregation of the information is the so-called *partial information content* $M(\varepsilon) := |\Phi'|/(n-1)$, where Φ' is the symbol sequence of alternating 1’s and $\bar{1}$’s, which is derived from Φ by removing all zeros and repeated symbols. Muñoz et al. (2015a) then suggest to use these two characteristics for computing the following five features:

1. *maximum information content* $H_{max} := \max_\varepsilon\{H(\varepsilon)\}$,
2. *settling sensitivity* $\varepsilon_s := \log_{10}(\min_\varepsilon\{\varepsilon : H(\varepsilon) < s\})$, with the default of s being 0.05 as suggested by Muñoz et al. (2015a),
3. $\varepsilon_{max} := \arg\max_\varepsilon\{H(\varepsilon)\}$,
4. *initial partial information* $M_0 := M(\varepsilon = 0)$,
5. *ratio of partial information sensitivity* $\varepsilon_r := \log_{10}(\max_\varepsilon\{\varepsilon : M(\varepsilon) > r \cdot M_0\})$, with the default of r being 0.5 as suggested by Muñoz et al. (2015a).

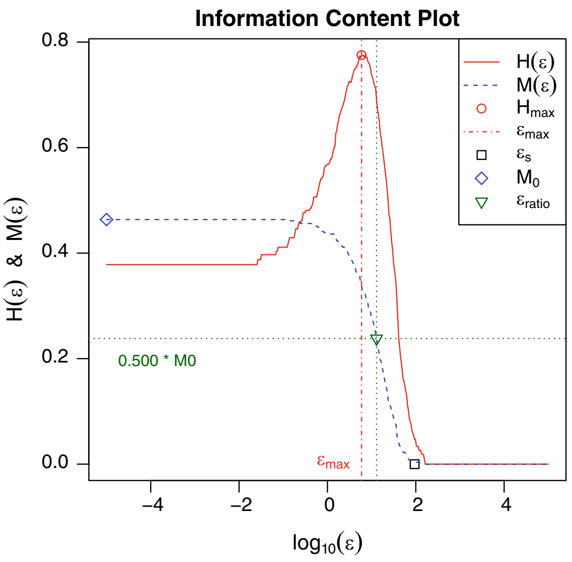


Fig. 7.8 *Information Content Plot* of a 2D version of *Gallagher's Gaussian 101-me Peaks* function. This approach analyzes the behavior of the tour along the points of the problem's initial design

The various characteristics and features described above can also be visualized within an *Information Content Plot* as exemplarily shown in Fig. 7.8.

Further Miscellaneous Features (40/3):
The remaining three feature sets provided within `flacco` are based on some very simple ideas. Note that there exists no (further) detailed description for these three feature sets as—according to the best knowledge of the author—none of them has been used for any (published) experiments. However, from our perspective they are worth to be included in future studies.

The simplest and quickest computable group of features are the 16 *basic* features, which provide rather obvious information of the initial design: (a) the number of observations and input variables, (b) the minimum and maximum of the lower and upper boundaries, objective values and number of blocks per dimension, (c) the number of filled cells and the total number of cells, and (d) a binary flag stating whether the objective function should be minimized.

The 14 *linear model* features fit a linear model—with the objective variable being the depending variable and all the remaining variables as explaining variables—within each cell of the feature object and aggregate the coefficient vectors across all the cells. That is, we compute (a) the length of the average coefficient vector, (b) the correlation of the coefficient vectors, (c) the ratio of maximum and minimum, as well as the arithmetic mean of the standard deviation of the (non-intercept) coefficients, (d) the previous features based on coefficient vectors that were a priori normalized per cell, (e) the arithmetic mean and standard deviation of the lengths of the coefficient vectors (across the cells), and (f) the arithmetic mean and standard deviation of the ratio of biggest and smallest (non-intercept) coefficients per cell.

The remaining ten features extract information from a *principal component* analysis, which is performed on the initial design—both, including and excluding the objective values—and that are based on the covariance, as well as correlation matrix, respectively. For each of these four combinations, the features measure the proportion of principal components that are needed to explain a predefined percentage (default: 0.9) of the data's variance, as well as the percentage of variation that is explained by the first principal component.

Further information on all implemented features, especially more detailed information on each of its configurable parameters, is given within the package's documentation. Note that in contrast to the original implementation of most of the feature sets the majority of parameters, e.g., the classifiers used by the level set features, or the ratio for the partial information sensitivity (information content), are configurable. While for reasons of conveniency, the default values are set according to their original publications, they can easily be changed using the `control` argument within the functions `calculateFeatureSet` or `calculateFeatures`, respectively. An example for this is given at the end of Sect. 7.3.

Note that each feature set can be computed for unconstrained, as well as box-constrained optimization problems. Upon creation of the feature object, `R` asserts that each point from the initial sample `X` lies within the defined `lower` and `upper` boundaries. Hence, if one intends to consider unconstrained problems, one should set the boundaries to `-Inf` or `Inf`, respectively. When dealing with box-constrained problems, one has to take a look at the feature sets themselves. Given that 14 out of the 17 feature sets do not perform additional function evaluations at all, none of them can violate any of the constraints. This also holds for the remaining three feature sets. The *convexity* features always evaluate (additional) points that are located in the center of two already existing (and thereby feasible) points. Consequently, each of them has to be located within the box-constraints as well. In case of the *local search* features, the box-constraints are also respected as long as one uses the default local search algorithm (L-BFGS-B), which was designed to optimize bound-constrained optimization problems. Finally, the *curvature* features do not violate the constraints either as the internal functions, which are used for estimating the gradient and Hessian of the function, were adapted in such a way that they always estimate them by evaluating points within the box-constraints—even in case the points are located close to or even exactly on the border of the box.

As described within the previous two sections, numerous researchers have already developed feature sets and many of them also shared their source code, enabling the creation of this wide collection of landscape features in the first place. By developing `flacco` on a publicly accessible platform,[7] other `R`-users may easily contribute (further feature sets) to the package. Aside from using the most recent (development) version of the package, one can also use its stable release from CRAN.[8] Moreover, the package repository on GitHub also provides a link to the corresponding online

[7]The development version is available on GitHub: https://github.com/kerschke/flacco.

[8]The stable release is published on CRAN: https://cran.r-project.org/package=flacco.

tutorial,[9] the platform-independent web-application[10] (further details are given in Sect. 7.5) and an issue tracker, where one could report bugs, provide feedback or suggest the integration of further feature sets.

7.3 Exemplary Feature Computation

In the following, the usage of `flacco` will exemplarily be presented on a well-known optimization problem, namely *Gallagher's Gaussian 101-me Peaks* (Hansen et al. 2009). It is the 21st out of the 24 artificially designed, continuous single-objective optimization problems from the *Black-Box Optimization Benchmark* (BBOB, Hansen et al. 2010). Within this benchmark, it belongs to a group of five multimodal problem instances with a weak global structure. Each of the 24 function classes from BBOB can be seen as a specific problem generator, whose problems are identical up to rotation, shifts and scaling. Therefore the exact instance has to be defined by a function ID (`fid`) and an instance ID (`iid`). For this exemplary presentation, we will use the second instance and generate it using `makeBBOBFunction` from the `R`-package `smoof` (Bossek 2017). The resulting landscape is depicted as a three-dimensional perspective plot in Fig. 7.9 and as a contour plot in Fig. 7.10.

In a first step, one needs to install the package along with all its dependencies, load it into the workspace and afterward generate the input data—i.e., a (hopefully) representative sample `X` of observations from the decision space along with their corresponding objective values `y`.

```
> install.packages("flacco", dependencies = TRUE)
> library("flacco")
> f = smoof::makeBBOBFunction(dimension = 2, fid = 21, iid = 2)
> ctrl = list(init_sample.lower = -5, init_sample.upper = 5)
> X = createInitialSample(n.obs = 100, dim = 2, control = ctrl)
> y = apply(X, 1, f)
```

The code above would also work without explicitly specifying the lower and upper boundaries (within `ctrl`), but in that case, the decision sample would automatically be generated within $[0, 1] \times [0, 1]$ rather than within the (for BBOB) common box-constraints of $[-5, 5] \times [-5, 5]$. In a next step, this input data is used (eventually in combination with further parameters such as the number of blocks per dimension) to create the feature object as already schematized in Fig. 7.3. The following `R`-code shows how this could be achieved.

```
> feat.object = createFeatureObject(X = X, y = y, fun = f,
+   blocks = c(8, 5))
```

As stated earlier, a feature object is the fundamental object for the majority of computations performed by `flacco` and thus, it obviously has to store quite a lot of

[9]Link to the package's tutorial: http://kerschke.github.io/flacco-tutorial/.

[10]Link to the package's GUI: https://flacco.shinyapps.io/flacco/.

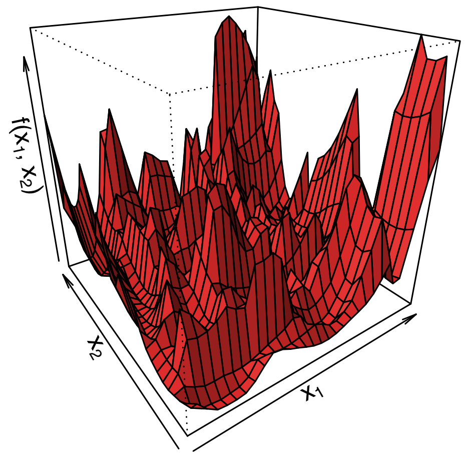


Fig. 7.9 3D-Perspective plot of *Gallagher's Gaussian 101-me Peaks* function

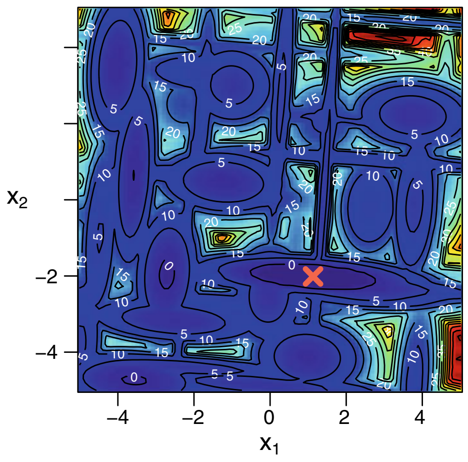


Fig. 7.10 Contour plot of *Gallagher's Gaussian 101-me Peaks* function. The red cross marks the global optimum and the coloring of the plot represents the objective values

information—such as the number of observations and (input) variables, the names and boundaries of the variables, the amount of empty and nonempty cells, etc. Parts of that information can easily be printed to the console by calling `print` on the feature object.[11]

[11]Note that the shown numbers, especially the ones for the number of observations per cell, might be different on your machine, as the initial sample is drawn randomly.

```
> print(feat.object)

  Feature Object:
  - Number of Observations: 100
  - Number of Variables: 2
  - Lower Boundaries: -5.00e+00, -5.00e+00
  - Upper Boundaries: 5.00e+00, 5.00e+00
  - Name of Variables: x1, x2
  - Optimization Problem: minimize y
  - Function to be Optimized: smoof-function (BBOB_2_21_2)
  - Number of Cells per Dimension: 8, 5
  - Size of Cells per Dimension: 1.25, 2.00
  - Number of Cells:
    - total: 40
    - non-empty: 36 (90.00%)
    - empty: 4 (10.00%)
  - Average Number of Observations per Cell:
    - total: 2.50
    - non-empty: 2.78
```

Given the feature object, one can easily calculate any of the 17 feature sets that were introduced in Sect. 7.2. Specific feature sets can for instance—exemplarily shown for the *angle*, *dispersion* and *nearest better clustering* features—be computed as shown in the code below.

```
> angle.features = calculateFeatureSet(
+   feat.object = feat.object, set = "cm_angle")
> dispersion.features = calculateFeatureSet(
+   feat.object = feat.object, set = "disp")
> nbc.features = calculateFeatureSet(
+   feat.object = feat.object, set = "nbc")
```

Alternatively, one can compute all implemented features simultaneously via:

```
> all.features = calculateFeatures(feat.object)
```

Each of these calculations results in a list of (mostly numeric, i.e., real-valued) features. In order to avoid errors due to possibly similar feature names among different feature sets, each feature inherits the abbreviation of its feature set's name as a prefix, e.g., `cm_angle` for the *cell mapping angle* features or `disp` for the *dispersion* features. Using the results from the previous example, the output for the calculation of the seven *nearest better clustering* features could look like this:

```
> str(nbc.features)

  List of 7
  $ nbc.nn_nb.sd_ratio       : num 0.382
  $ nbc.nn_nb.mean_ratio     : num 0.59
  $ nbc.nn_nb.cor            : num 0.264
  $ nbc.dist_ratio.coeff_var: num 0.391
  $ nbc.nb_fitness.cor       : num -0.591
  $ nbc.costs_fun_evals      : int 0
  $ nbc.costs_runtime        : num 0.022
```

At this point, the authors would like to emphasize that **one should not try to interpret the numerical feature values** on their own as the majority of them simply are not intuitively understandable. Instead, these numbers should rather be used to distinguish between different problems—usually in an automated fashion, e.g., by means of a machine learning algorithm.

Also, it is important to note that features, which belong to the same feature set, are based on the same idea and only differ in the way they aggregate those ideas. For instance, the *nearest better clustering* features are based on the same distance sets—the distances to their nearest neighbors and nearest better neighbors. However, the features differ in the way they aggregate those distance sets, e.g., by computing the ratio of their (i) standard deviations or (ii) arithmetic means. As the underlying approaches are the bottleneck of each feature set, computing single features on their own would not be very beneficial and therefore has not been implemented in this package. Therefore, given the fact that multiple features share a common idea, which is usually the expensive part of the feature computation—for this group of features it would be the computation of the two distance sets—they will be computed together as an entire feature set instead of computing each feature on its own.

Figure 7.11 shows the results of a so-called "microbenchmark" (Mersmann 2014) on the chosen BBOB instance. Such a benchmark allows the comparison of the (logarithmic) runtimes across the different feature sets. As one can see, the time for calculating a specific feature set heavily depends on the chosen feature set. Although the majority of feature sets can be computed in less than half a second (the *basic* features even in less than a millisecond), three feature sets—namely *curvature*, *level set* and *local search*—usually needed between one and two seconds. And although two seconds might not sound that bad, it can have a huge impact on the required resources when computing all feature sets across thousands or even millions of instances. The amount of required time might even increase when considering the fact that some of the features are stochastic. In such a case it is strongly recommended to calculate each of those feature sets multiple times (on the same problem instance) in order to capture the variance of these features.

Note that the current definition of the feature object also contains the actual *Gallagher's Gaussian 101-me Peaks* function itself—it was added to the feature object by setting the parameter `fun = f` within the function `createFeatureObject`. Without that information it would have been impossible to calculate feature sets,

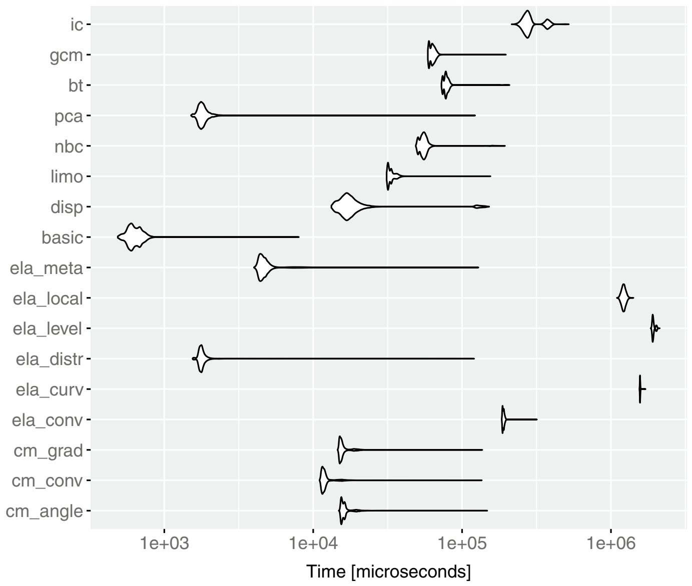


Fig. 7.11 Violin plots of a microbenchmark, measuring the logarithmic runtimes (in microseconds) of each feature set based on 1,000 replications on a two-dimensional version of the *Gallagher's Gaussian 101-me Peaks* function

which require additional function evaluations, i.e., the *convexity*, *curvature* and *local search* features of the classical ELA features. Similarly, the *barrier tree*, *cell mapping* and *general cell mapping* features strongly depend on the value of the `blocks` argument, which defines the number of blocks per dimension when representing the decision space as a grid of cells.[12]

As mentioned in Sect. 7.2, the majority of feature sets uses some parameters, all of them having certain default settings. However, as different landscapes/optimization problems might require different configurations of these parameters, the package allows its users to change the default parameters by setting their parameter configuration within the `control` argument of the functions `calculateFeatures` or `calculateFeatureSet`, respectively. Similarly to the naming convention of the features themselves, the names of parameters use the abbreviation of the corresponding feature set as a prefix. This way, one can store all the preferred configurations within a single control argument and does not have to deal with a separate control argument for each feature set. In order to illustrate the usage of the control argument, let's assume one wants to solve the following two tasks:

[12]The *barrier tree* features can only be computed if the total number of cells is at least two and the *cell mapping convexity* features require at least three blocks per dimension.

1. Calculate the *dispersion* features for a problem, where the Manhattan distance (also known as city-block distance or L_1-norm) is more reasonable than the (default) Euclidean distance.
2. Calculate the *General Cell Mapping* features for two instead of the usual three GCM approaches (i.e., `"min"` = best value per cell, `"mean"` = average value per cell and `"near"` = closest value to a cell center).

Due to the availability of a shared control parameter, these adaptations can easily be performed with a minimum of additional code:

```
> ctrl = list(
+   disp.dist_method = "manhattan",
+   gcm.approaches = c("min", "near")
+ )
> dispersion.features = calculateFeatureSet(
+   feat.object = feat.object, set = "disp", control = ctrl)
> gcm.features = calculateFeatureSet(
+   feat.object = feat.object, set = "gcm", control = ctrl)
```

This control parameter also works when computing all features simultaneously:

```
> all.features = calculateFeatures(feat.object, control = ctrl)
```

A detailed overview of all the configurable parameters is available within the documentation of the previous functions and can for instance be accessed via the command `?calculateFeatures`.

7.4 Visualization Techniques

In addition to the features themselves, `flacco` provides a handful of visualization techniques allowing the user to get a better understanding of the features. The function `plotCellMapping` produces a plot of the discretized grid of cells, representing the problem's landscape when using the cell mapping approach. Figure 7.6, which was already presented in Sect. 7.2, shows two of these cell mapping plots—the left one represents each cell by the smallest fitness value of its respective points, and the right plot represents each cell by the average of the fitness values of the respective cell's sample points. In each of those plots, the *attractor cells*, i.e., cells containing the sample's local optima, are filled black. The *uncertain cells*, i.e., cells that transition to multiple attractors, are colored gray, whereas the *certain cells* are colored according to their *basin of attraction*, i.e., all cells which transition towards the same attractor (and no other one) share the same color. Additionally, the figure highlights the attracting cells of each cell via arrows that point towards the corresponding attractor(s). Note that the length of the arrows is proportional to the transition probabilities.

Given the feature object from the example of the previous section, i.e., the one that is based on *Gallagher's Gaussian 101-me Peaks* function, the two cell mapping plots can be produced with the following code:

```
> plotCellMapping(feat.object,
+   control = list(gcm.approach = "min"))
> plotCellMapping(feat.object,
+   control = list(gcm.approach = "mean"))
```

Analog to the cell mapping plots, one can visualize the barrier trees of a problem. They can either be plotted in 2D as a layer on top of a cell mapping (per default without any arrows) or in 3D as a layer on top of a perspective/surface plot of the discretized decision space. Using the given feature object, the code for creating the plots from Fig. 7.7 (also already shown in Sect. 7.2) would look like this:

```
> plotBarrierTree2D(feat.object)
> plotBarrierTree3D(feat.object)
```

The representation of the tree itself is in both cases very similar. Each of the trees begins with its root, depicted by a red triangle. Starting from that point, there will usually be two or more branches, pointing either towards a saddle point (depicted by a non-filled diamond)—a node that connects multiple basins of attraction—or towards a leaf (filled circles) of the tree, indicating a local optimum. Branches belonging to the same level (i.e., having the same number of predecessors on their path up to the root) of the tree are colored identically. Note that all of the aforementioned points, i.e., root, saddle points and leaves, belong to distinct cells. The corresponding (unique) cell IDs are given by the numbers next to the points.

The last plot, which is directly related to the visualization of a feature set, is the so-called *information content plot*, which was shown in Fig. 7.8 in Sect. 7.2. It depicts the logarithm of the *information sensitivity* ε against the *(partial) information content*. Following these two curves—the solid red line representing the *information content* $H(\varepsilon)$, and the dashed blue line representing the *partial information content* $M(\varepsilon)$—one can (more or less) easily derive the features from the plot. The blue diamond at the very left represents the *initial partial information* M_0, the red circle on top of the solid, red line shows the *maximum information content* H_{max}, the green triangle indicates the *ratio of partial information sensitivity* $\varepsilon_{0.5}$ (with a ratio of $r = 0.5$) and the black square marks the *settling sensitivity* ε_s (with $s = 0.05$). Further details on the interpretation of this plot and the related features can be found in Muñoz et al. (2012, 2015a). In accordance to the code of the previous visualization techniques, such an information content plot can be created by the following command:

```
> plotInformationContent(feat.object)
```

In addition to the previously introduced visualization methods, `flacco` provides another plot function. However, in contrast to the previous ones, the *Feature Importance Plot* (cf. Fig. 7.12) does not visualize a specific feature set. Instead, it can be used to assess the importance of several features during a feature selection process of a machine learning (e.g., classification or regression) task. Given the high amount of features, provided by this package, it is very likely that many of them are redundant

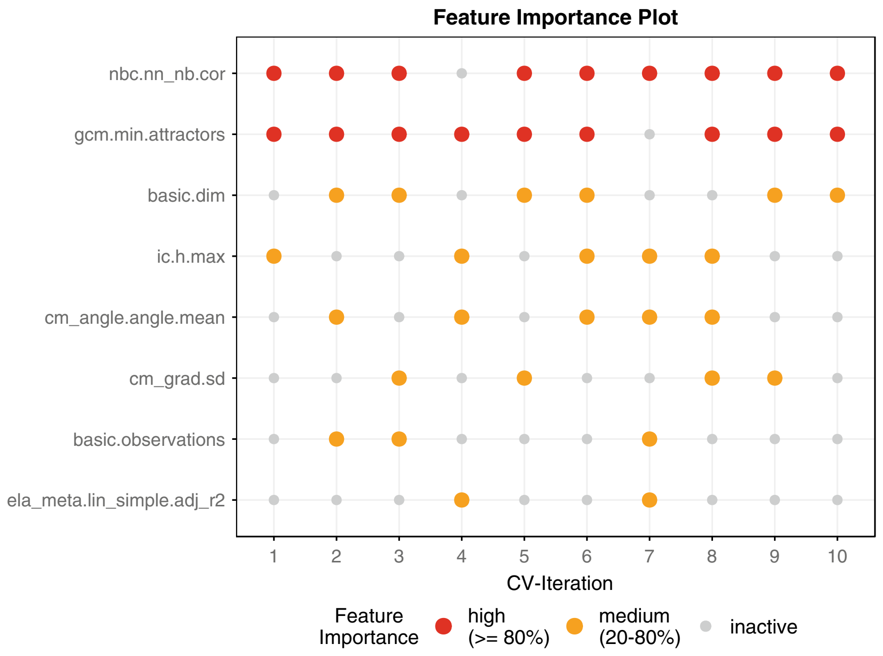


Fig. 7.12 Plot of the feature importance as derived from a 10-fold cross-validated feature selection

when training a (classification or regression) model and therefore, a feature selection strategy would be useful.

The scenario of Fig. 7.12 shows the (artificially created) result of such a feature selection after performing a tenfold cross-validation using the R-package `mlr` (Bischl et al. 2016).[13] Such a resampling strategy is useful in order to assess whether a certain feature was selected because of its importance to the model or more or less just by chance. The red points show whether an important feature—i.e., a feature, which has been selected in the majority of iterations (default: $\geq 80\%$)—was selected within a specific iteration (= fold of the cross-validation). Less important features are depicted as orange points. The only information that is required to create such a *Feature Importance Plot* is a list, whose elements (one per fold / iteration) are vectors of character strings. In the illustrated example, this list consists of ten elements (one per fold) with the first one being a character vector that contains exactly the three features `"gcm.min.attractors"`, `"nbc.nn_nb.cor"` and `"ic.h.max"`. The entire list that leads to the plot from Fig. 7.12 (denoted as `list.of.features`) would look like this:

[13]A more detailed step-by-step example can be found in the documentation of the respective `flacco`-function `plotFeatureImportancePlot`.

```
> str(list.of.features, vec.len = 2L)

  List of 10
   $ : chr [1:3] "gcm.min.attractors" "ic.h.max" ...
   $ : chr [1:5] "gcm.min.attractors" "nbc.nn_nb.cor" ...
   $ : chr [1:5] "gcm.min.attractors" "nbc.nn_nb.cor" ...
   $ : chr [1:4] "gcm.min.attractors" "ic.h.max" ...
   $ : chr [1:4] "gcm.min.attractors" "nbc.nn_nb.cor" ...
   $ : chr [1:5] "gcm.min.attractors" "ic.h.max" ...
   $ : chr [1:5] "ic.h.max" "nbc.nn_nb.cor" ...
   $ : chr [1:5] "gcm.min.attractors" "ic.h.max" ...
   $ : chr [1:4] "gcm.min.attractors" "nbc.nn_nb.cor" ...
   $ : chr [1:3] "gcm.min.attractors" "nbc.nn_nb.cor" ...
```

Given such a list of features, the *Feature Importance Plot* can be generated with the following command:

```
> ggplotFeatureImportance(list.of.features)
```

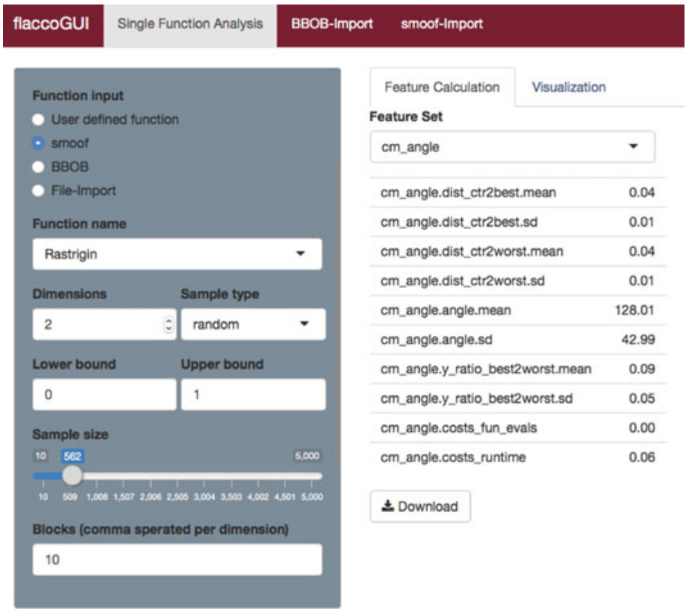
Similarly to the feature computations, which were described in Sect. 7.3, each of the plots can be modified according to a user's needs by making use of the `control` parameter within each of the plot functions. Further details on the possible configurations of each of the visualization techniques can be found in their documentation.

7.5 Graphical User Interface

Within the previous sections, we have introduced a wide collection of feature sets that were previously implemented in different programming languages or at least in different packages. In addition, we have shown how one can use `flacco` to compute ELA features. While this is beneficial for researchers who are familiar with `R`, it comes with the drawback that researchers who are not familiar with `R` are left out. Therefore, Hanster and Kerschke (2017) implemented a graphical user interface (GUI) for the package, which can either be started from within `R`, or—which is probably more appealing to non-`R`-users—as a platform-independent web-application.

The GUI was implemented using the `R`-package `shiny` (Chang et al. 2016) and according to its developers, `shiny` "*makes it incredibly easy to build interactive web applications with* `R`" and its "*extensive prebuilt widgets make it possible to build beautiful, responsive, and powerful applications with minimal effort.*"

If one wants to use the GUI-version that is integrated within `flacco` (version 1.5 or higher), one first needs to install `flacco` from CRAN (https://cran.r-project.org/package=flacco) and load it into the workspace of `R`.



Function input
- User defined function
- [✓ smoof]
- BBOB
- File-Import

Function name: Rastrigin

Dimensions: 2

Sample type: random

Lower bound: 0

Upper bound: 1

Sample size: 562 (10 – 5,000)

Blocks (comma sperated per dimension): 10

Feature Set: cm_angle

cm_angle.dist_ctr2best.mean	0.04
cm_angle.dist_ctr2best.sd	0.01
cm_angle.dist_ctr2worst.mean	0.04
cm_angle.dist_ctr2worst.sd	0.01
cm_angle.angle.mean	128.01
cm_angle.angle.sd	42.99
cm_angle.y_ratio_best2worst.mean	0.09
cm_angle.y_ratio_best2worst.sd	0.05
cm_angle.costs_fun_evals	0.00
cm_angle.costs_runtime	0.06

Fig. 7.13 Screenshot of the GUI, after computing the *cell mapping angle* features for a two-dimensional version of the *Rastrigin* function as implemented in `smoof`

```
> install.packages("flacco", dependencies = TRUE)
> library("flacco")
```

Afterwards it only takes the following line of code to start the application:

```
> runFlaccoGUI()
```

If one rather prefers to use the platform-independent GUI, one can use the web application, which is hosted on https://flacco.shinyapps.io/flacco/.

Once started, the application shows a bar on the top, where the user can select between the following three options (as depicted in Fig. 7.13): “Single Function Analysis”, “BBOB-Import” and “Smoof-Import”. Selecting the first tab, the application will show two windows: (1) a box (colored in gray) on the left, which inquires all the information that is needed for creating the feature object, i.e., the underlying function, the problem dimension (= number of input variables), the boundaries, the size and sampling type (latin hypercube or random uniform sampling) of the initial sample and the number of cells per dimension, and (2) a screen for the output on the right, where one can either compute the numerical landscape features under the tab “Feature Calculation” (as shown in Fig. 7.13) or create a related plot under the tab “Visualization” (as shown in Fig. 7.14).

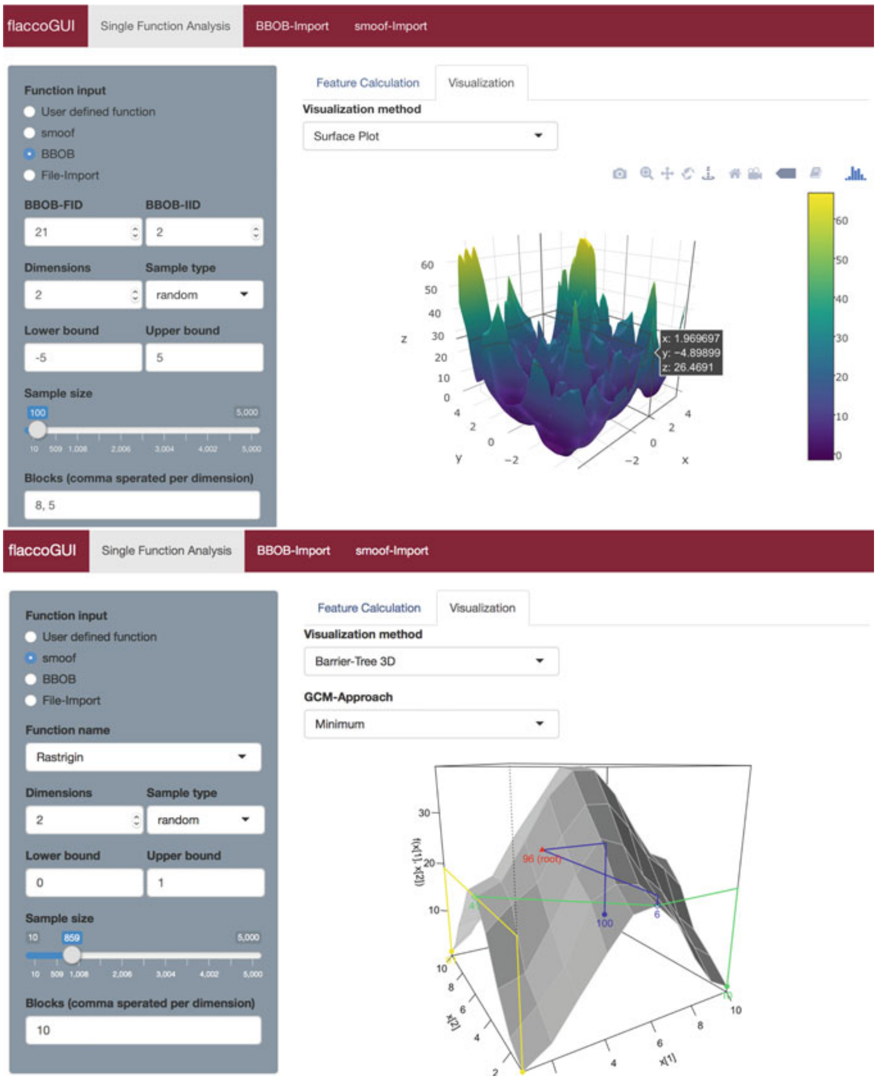
For the definition of the underlying function (within the gray box), the user has the following options: (1) providing a user-defined function by entering the corresponding R expression, e.g., `sum(x^2)`, into a text box, (2) selecting one of the single-objective problems provided by `smoof` from a drop-down menu, (3) defining a BBOB function (Hansen et al. 2010) via its function ID (FID) and instance ID (IID), or (4) ignoring the function input and just provide the initial design, i.e., the input variables and the corresponding objective value, by importing a CSV-file.

Immediately after the necessary information for the feature object is provided, the application will automatically produce the selected output on the right half of the screen. If one does not select anything (on the right half), the application will try to compute the *cell mapping angle* features, as this is the default selection at the start of the GUI. However, the user can select any other feature set, as well as the option "all features", from a drop-down menu. Once the computation of the features is completed, one can download the results as a CSV-file by clicking on the "Download"-button underneath the feature values.

Aside from the computation of the landscape features, the GUI also provides functionalities for generating multiple visualizations of the given feature object. Aside from the *cell mapping plots* and two- and three-dimensional *barrier trees*, which can only be created for two-dimensional problems, it is also able to produce *information content plots* (cf. Sect. 7.4). Note that the latter currently is the only plotting function that is able to visualize problems with more than two input variables. In addition to the aforementioned plots, the application also enables the visualization of the function itself—if applicable—by means of an interactive *function plot* (for one-dimensional problems), as well as two-dimensional *contour* or three-dimensional *surface plots* (for two-dimensional problems). The interactivity of these plots, such as the one that is shown in the upper image of Fig. 7.14, was provided by the R-package `plotly` (Sievert et al. 2016) and (amongst others) enables to zoom in and out of the landscape, as well as to rotate or shift the latter.

While `flacco` itself allows the configuration of all of its parameters—i.e., the parameters used by any of the feature sets, as well as the ones related to the plotting functions—the GUI provides a simplified version of this. Hence, for the cell mapping and barrier tree plots, one can choose the approach for selecting the representative objective value per cell: (a) the best objective value among the samples from the cell (as shown in the lower image of Fig. 7.14), (b) the average of all objective values from the cell's sample points, or (c) the objective value of the observation that is located closest to the cell's center. In case of the information content plot, one can configure the range of the x-axis, i.e., the range of the (logarithmic) values of the information sensitivity parameter ε.

As already mentioned in Sect. 7.3, the authors highly recommend not to interpret single features on single optimization problems. Instead, one should rather compute several (sets of) landscape features across an entire benchmark of optimization problems. This way, they might be able to provide the information for distinguishing the benchmark's problems from each other and predicting a well-performing optimization algorithm per instance—cf. the *algorithm selection problem*, which was already mentioned as a potential use case in Sect. 7.1—or for adapting the parameters of such an optimizer to the underlying landscape.


flaccoGUI | Single Function Analysis | BBOB-Import | smoof-Import

Function input: User defined function / smoof / [✓ BBOB] / File-Import

BBOB-FID: 21 BBOB-IID: 2

Dimensions: 2 Sample type: random

Lower bound: -5 Upper bound: 5

Sample size: 100 (10 – 5,000)

Blocks (comma sperated per dimension): 8, 5

Feature Calculation | Visualization

Visualization method: Surface Plot

x: 1.969697
y: −4.89899
z: 26.4691

flaccoGUI | Single Function Analysis | BBOB-Import | smoof-Import

Function input: User defined function / [✓ smoof] / BBOB / File-Import

Function name: Rastrigin

Dimensions: 2 Sample type: random

Lower bound: 0 Upper bound: 1

Sample size: 859 (10 – 5,000)

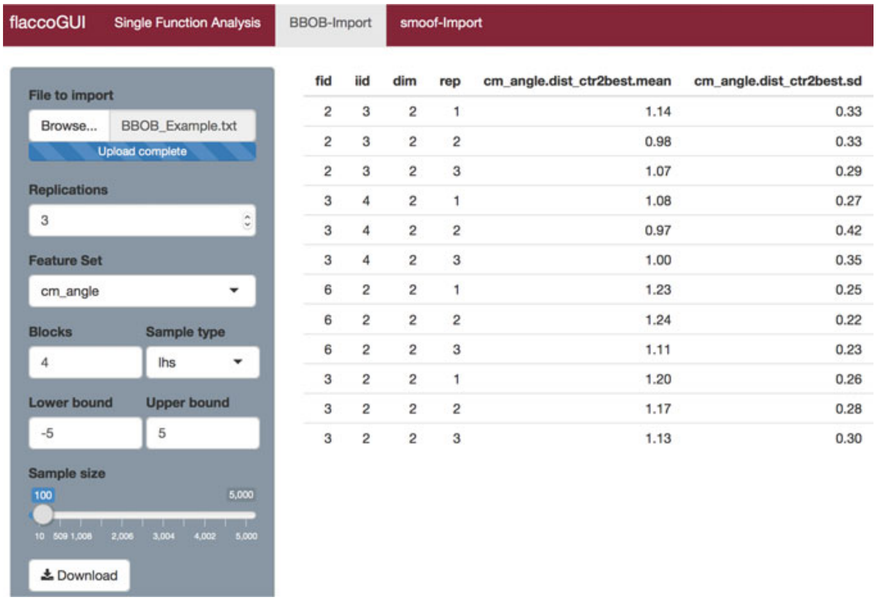
Blocks (comma sperated per dimension): 10

Feature Calculation | Visualization

Visualization method: Barrier-Tree 3D

GCM-Approach: Minimum

Fig. 7.14 Exemplary screenshots of two visualization techniques as provided by the GUI and shown for two different optimization problems. Top: An interactive surface plot of an instance of *Gallagher's 101-me Peaks* function (second instance of the 21st BBOB problem), which allows to zoom in and out of the landscape, as well as to rotate or shift it. Bottom: A 3D representation of the barrier tree for an instance of the *Rastrigin* function as implemented in `smoof`



File to import: BBOB_Example.txt

Replications: 3

Feature Set: cm_angle

Blocks: 4

Sample type: lhs

Lower bound: -5

Upper bound: 5

Sample size: 100 (10, 509, 1,008, 2,006, 3,004, 4,002, 5,000)

fid	iid	dim	rep	cm_angle.dist_ctr2best.mean	cm_angle.dist_ctr2best.sd
2	3	2	1	1.14	0.33
2	3	2	2	0.98	0.33
2	3	2	3	1.07	0.29
3	4	2	1	1.08	0.27
3	4	2	2	0.97	0.42
3	4	2	3	1.00	0.35
6	2	2	1	1.23	0.25
6	2	2	2	1.24	0.22
6	2	2	3	1.11	0.23
3	2	2	1	1.20	0.26
3	2	2	2	1.17	0.28
3	2	2	3	1.13	0.30

Fig. 7.15 Screenshot of the GUI, after computing the *cell mapping angle* features for four different BBOB instances with three replications each. Note that the remaining eight features from this set were computed as well and were simply cut off for better visibility

In order to facilitate such experiments, our application also provides functionalities for computing a specific feature set (or even all feature sets) simultaneously on multiple instances of a specific problem class—such as the BBOB functions or any of the other single-objective optimization problems provided by `smoof`. Figure 7.15 shows an exemplary screenshot of how such a batch computation could look like for the BBOB problems. In a first step, one has to upload a CSV-file consisting of three columns—the function ID (FID), instance ID (IID) and problem dimension (dim)—and one row per problem instance. Next, one defines the desired number of replications, i.e., how often should the features be computed per instance.[14] Afterward, one selects the feature set that is supposed to be computed, defines number of blocks and boundaries per dimension, as well as the desired sample type and size. Note that, depending on the size of the benchmark, i.e., the number of instances and replications, the number and dimensions of the initial sample, as well as the complexity of the chosen feature set, the computation of the features might take a while. Furthermore, please note that the entire application is reactive. That is, once all the required fields provide some information, the computation of the features will start and each time the user changes the information within a field, the computation will automatically restart. Therefore, the authors recommend to first configure

[14]As many of the features are stochastic, it is highly recommended to compute the features multiple(= at least 5 to 10) times.

all the parameters prior to uploading the CSV-file as the latter is the only missing piece of information at the start of the application. Once the feature computation was successfully completed, one can download the computed features/feature sets as a CSV-file by clicking on the “Download”-Button.

Analogously to the batch-wise feature computation on BBOB instances, one can also perform batch-wise feature computation for any of the other single-objective problems that are implemented in `smoof`. For this purpose, one simply has to go to the tab labeled “smoof-Import” and perform the same steps as described before. The only difference is, that the CSV-file with the parameters now only consists of a single column (with the problem dimension).

7.6 Summary

The `R`-package `flacco` provides a collection of numerous features for exploring and characterizing the landscape of continuous, single-objective (optimization) problems by means of numerical values, which can be computed based on rather small samples from the decision space. Having this wide variety of feature sets bundled within a single package simplifies further researches significantly—especially in the fields of algorithm selection and configuration—as one does not have to run experiments across multiple platforms. In addition, the package comes with different visualization strategies, which can be used for analyzing and illustrating certain feature sets, leading towards an improved understanding of the features.

This framework can be meaningfully combined with other `R`-packages such as `mlr` (Bischl et al. 2016) or `smoof` (Bossek 2017) and thereby enables the construction of better-performing optimization algorithms (which use the features to adapt to the problem at hand) or even algorithm selectors. Hence, `flacco` is a useful toolbox when (i) performing exploratory landscape analysis, and (ii) working towards algorithm selection models of single-objective, continuous optimization problems. Therefore, the package can be considered as a solid foundation for further researches—as for instance performed by members of the COSEAL network[15]—allowing to get a better understanding of algorithm selection scenarios and the employed optimization algorithms.

Lastly, `flacco` also comes with a graphical user interface, which can either be run from within `R`, or alternatively as a platform-independent standalone web-application. Therefore, even people who are not familiar with `R` have the chance to benefit of the functionalities of this package—especially from the computation of numerous landscape features—without the burden of interfacing or re-implementing identical features in a different programming language.[16]

[15]COSEAL is an international group of researchers with a focus on the **Co**nfiguration and **Se**lection of **Al**gorithms, cf. http://www.coseal.net/.

[16]The *European Center of Information Systems* (ERCIS) is an international network in the field of Information Systems, cf. https://www.ercis.org/.

Acknowledgements We acknowledge support by the ERCIS and thank Carlos Hernández (CINVESTAV, Mexico), Jan Dageförde, as well as Christian Hanster (University of Münster, Germany) for their valuable contributions to `flacco` and its GUI.

References

Abell, T., Malitsky, Y., & Tierney, K. (2013). Features for exploiting black-box optimization problem structure. In *Learning and intelligent optimization* (pp. 30–36). Berlin: Springer.

Beachkofski, B. K., & Grandhi, R. V. (2002). Improved distributed hypercube sampling. In *43rd AIAA/ASME/ASCE/AHS/ASC structures, structural dynamics, and materials conference*.

Bischl, B., Lang, M., Kotthoff, L., Schiffner, J., Richter, J., & Studerus, E., et al. (2016). mlr: Machine Learning in R. *Journal of Machine Learning Research*, *17*(170), 1–5. `R`-package version 2.10.

Bischl, B., Mersmann, O., Trautmann, H., & Preuss, M. (2012). Algorithm selection based on exploratory landscape analysis and cost-sensitive learning. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, GECCO '12* (pp. 313–320). New York: ACM.

Bossek, J. (2017). smoof: Single-and multi-objective optimization test functions. *The R Journal*. https://journal.r-project.org/archive/2017/RJ-2017-004/index.html.

Byrd, R. H., Peihuang, L., Nocedal, J., & Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, *16*(5), 1190–1208.

Chang, W., Cheng, J., Allaire, J. J., Xie, Y., & McPherson, J. (2016). *Shiny: Web application framework for R*. `R`-package version 0.14.1.

Christoph, F., Hofacker, I. L., Stadler, P. F., & Wolfinger, M. T. (2002). Barrier trees of degenerate landscapes. *Zeitschrift für Physikalische Chemie International Journal of Research in Physical Chemistry and Chemical Physics 216*(2/2002), 155.

Daolio, F., Liefooghe, A., Verel, S., Aguirre, H., & Tanaka, K. (2016). Problem features versus algorithm performance on rugged multi-objective combinatorial fitness landscapes. *Evolutionary Computation*.

Friedman, J. H. (1997). On bias, variance, 0/1–loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery*, *1*(1), 55–77.

Guido VanRossum and The Python Development Team. (2015). *The Python Language Reference–Release 3.5.0*. Python Software Foundation.

Hansen, N., Auger, A., Finck, S., & Ros, R. (2010). Real-parameter black-box optimization benchmarking 2010: experimental setup. Technical Report RR-7215, INRIA.

Hansen, N., Finck, S., Ros, R., & Auger, A. (2009). Real-parameter black-box optimization benchmarking 2009: noiseless functions definitions. Technical Report RR-6829, INRIA.

Hanster, C., & Kerschke, P. (2017). Flaccogui: Exploratory landscape analysis for everyone.

Hutter, F., Lin, X., Hoos, H. H., & Leyton-Brown, K. (2014). Algorithm runtime prediction: Methods and evaluation. *Journal of Artificial Intelligence*, *206*, 79–111.

Jobson, J. (2012). *Applied multivariate data analysis: Volume II: Categorical and multivariate methods*. Berlin: Springer.

Jones, T. (1995). *Evolutionary algorithms, fitness landscapes and search*. Ph.D. thesis, Citeseer.

Jones, T., & Forrest, S. (1995). Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th international conference on genetic algorithms* (pp. 184–192). Morgan Kaufmann Publishers Inc.

Kerschke, P. (2017). *Flacco: Feature-based landscape analysis of continuous and constrained optimization problems*. `R`-package version 1.6.

Kerschke, P., & Trautmann, H. (2016). The R-package FLACCO for exploratory landscape analysis with applications to multi-objective optimization problems. In *Proceedings of the IEEE congress on evolutionary computation (CEC)*. IEEE.

Kerschke, P., Preuss, M., Hernández, C., Schütze, O., Sun, J.- Q., Grimme, C., et al. (2014). Cell mapping techniques for exploratory landscape analysis. In *EVOLVE—A bridge between probability, set oriented numerics, and evolutionary computation V* (pp. 115–131). Berlin: Springer.

Kerschke, P., Preuss, M., Wessing, S., & Trautmann, H. (2015). Detecting funnel structures by means of exploratory landscape analysis. In *Proceedings of the 17th annual conference on genetic and evolutionary computation* (pp. 265–272). ACM.

Kerschke, P., Preuss, M., Wessing, S., & Trautmann, H. (2016). Low-budget exploratory landscape analysis on multiple peaks models. In *Proceedings of the 18th annual conference on genetic and evolutionary computation*. ACM.

Lunacek, M., & Whitley, D. (2006). The dispersion metric and the CMA evolution strategy. In *Proceedings of the 8th annual conference on genetic and evolutionary computation* (pp. 477–484). ACM.

Malan K. M., Oberholzer, J. F., & Engelbrecht, A. P. (2015). Characterising constrained continuous optimisation problems. In *Proceedings of the IEEE congress on evolutionary computation (CEC)* (pp. 1351–1358). IEEE.

Malan, K. M., & Engelbrecht, A. P. (2009). Quantifying ruggedness of continuous landscapes using entropy. In *Proceedings of the IEEE congress on evolutionary computation (CEC)* (pp. 1440–1447). IEEE.

MATLAB. (2013). *Version 8.2.0 (R2013b)*. The MathWorks Inc., Natick, Massachusetts.

McKay, M. D., Beckman, R. J., & Conover, W. J. (2000). A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, *42*(1), 55–61.

Mersmann, O. (2014). *Microbenchmark: Accurate timing functions*. R-package version 1.4-2.

Mersmann, O., Bischl, B., Trautmann, H., Preuss, M., Weihs, C., & Rudolph, G. (2011). Exploratory landscape analysis. In *Proceedings of the 13th annual conference on genetic and evolutionary computation, GECCO '11* (pp. 829–836). New York: ACM.

Mersmann, O., Bischl, B., Trautmann, H., Wagner, M., Bossek, J., & Neumann, F. (2013). A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem. *Annals of Mathematics and Artificial Intelligence*, *69*(2), 151–182.

Morgan, R., & Gallagher, M. (2015). Analysing and characterising optimization problems using length scale. *Soft Computing*, 1–18.

Müller, C. L., & Sbalzarini, I. F. (2011). Global characterization of the CEC 2005 fitness landscapes using fitness-distance analysis. In *Applications of evolutionary computation* (pp. 294–303). Berlin: Springer.

Muñoz, M. A., Kirley, M., & Halgamuge, S. K. (2012). Landscape characterization of numerical optimization problems using biased scattered data. In *2012 IEEE congress on evolutionary computation (CEC)* (pp. 1–8). IEEE.

Muñoz, M. A., Kirley, M., & Halgamuge, S. K. (2015a). Exploratory landscape analysis of continuous space optimization problems using information content. *IEEE transactions on evolutionary computation*, *19*(1), 74–87.

Muñoz, M. A., Sun, Y., Kirley, M., & Halgamuge, S. K. (2015b). Algorithm selection for black-box continuous optimization problems: A survey on methods and challenges. *Information Sciences*, *317*, 224–245.

Ochoa, G., Verel, S., Daolio, F., & Tomassini, M. (2014). Local optima networks: A new model of combinatorial fitness landscapes. In *Recent advances in the theory and application of fitness landscapes* (pp. 233–262). Berlin: Springer.

Pihera, J., & Musliu, N. (2014). Application of machine learning to algorithm selection for TSP. In *Proceedings of the IEEE 26th international conference on tools with artificial intelligence (ICTAI)*. IEEE press.

R Core Team. (2018). *R: A language and environment for statistical computing*. R foundation for statistical computing. Vienna, Austria.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Shirakawa, S., & Nagao, T. (2016). Bag of local landscape features for fitness landscape analysis. *Soft Computing*, *20*(10), 3787–3802.

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., et al. (2016). *Plotly: Create interactive web graphics via 'plotly.js'*. `R`-package version 4.5.6.