Package 'SPOT'

May 17, 2019

License GPL (>= 2)

Title Sequential Parameter Optimization Toolbox

Type Package

LazyLoad yes

LazyData true

Description A set of tools for model based optimization and tuning of algorithms. It includes surrogate models, optimizers and design of experiment approaches. The main interface is spot, which uses sequentially updated surrogate models for the purpose of efficient optimization. The main goal is to ease the burden of objective function evaluations, when a single evaluation requires a significant amount of resources.

Version 2.0.4

Date 2019-05-17

Depends R (>= 3.0.0)

Imports randomForest, stats, utils, graphics, grDevices, MASS, DEoptim, rgenoud, plotly, rsm, nloptr

RoxygenNote 6.1.1

Suggests testthat

NeedsCompilation no

Author Thomas Bartz-Beielstein [aut],
Joerg Stork [aut],
Martin Zaefferer [aut, cre],
Margarita Rebolledo [ctb],
Christian Lasarczyk [ctb],
Joerg Ziegenhirt [ctb],
Wolfgang Konen [ctb],
Oliver Flasch [ctb],
Patrick Koch [ctb],
Martina Friese [ctb],
Lorenzo Gentile [ctb],
Frederik Rehbach [ctb]

Maintainer Martin Zaefferer <martin.zaefferer@gmx.de>

Repository CRAN

Date/Publication 2019-05-17 17:50:07 UTC

R topics documented:

`SPOT-package` *Sequential Parameter Optimization Toolbox*

Description

Sequential Parameter Optimization Toolbox

Details

SPOT uses a combination statistic models and optimization algorithms for the purpose of parameter optimization. Design of Experiment methods are employed to generate an initial set of candidate solutions, which are evaluated with a user-provided objective function. The resulting data is used to fit a model, which in turn is subject to an optimization algorithm, to find the most promising candidate solution(s). These are again evaluated, after which the model is updated with the new results. This sequential procedure of modeling, optimization and evaluation is iterated until the evaluation budget is exhausted.

Note, that versions >= 2.0.1 of the package are a complete rewrite of the interfaces and conventions in SPOT. The rewritten SPOT package aims to improve the following issues of the older package:
- A more modular architecture is provided, that allows the user to easily customize parts of the SPO procedure.
- Core functions for modeling and optimization use interfaces more similar to algorithms from other packages / core-R, hence making them easier accessible for new users. Also, these can now be more easily used separately from the main SPO approach, e.g., only for modeling.
- Reducing the unnecessarily large number of choices and parameters.
- Removal of extremely rarely used / un-used features, to reduce overall complexity of the package.
- Improving documentation and accessibility in general.
- Speed-up of frequently used procedures.
We appreciate feedback about any bugs or other issues with the package. Feel free to send feedback by mail to the maintainer.

Package:	SPOT
Type:	Package
Version:	2.0.4
Date:	2019-05-17
License:	GPL (>= 2)
LazyLoad:	yes

Acknowledgments

This work has been partially supported by the Federal Ministry of Education and Research (BMBF) under the grants CIMO (FKZ 17002X11) and MCIOP (FKZ 17N0311).

Maintainer

Martin Zaefferer <martin.zaefferer@gmx.de>

Author(s)

Thomas Bartz-Beielstein <thomas.bartz-beielstein@th-koeln.de>, Joerg Stork, Martin Zaefferer (Maintainer, <martin.zaefferer@gmx.de>) with contributions from: C. Lasarczyk, M. Rebolledo, J. Ziegenhirt, W. Konen, O. Flasch, P. Koch, M. Friese, L. Gentile, F. Rehbach.

See Also

Main interface function is spot.

`buildEnsembleStack` *Ensemble: Stacking*

Description

Generates an ensemble of surrogate models with stacking (stacked generalization).

Usage

```
buildEnsembleStack(x, y, control = list())
```

Arguments

`x`	design matrix (sample locations), rows for each sample, columns for each variable.
`y`	vector of observations at `x`
`control`	(list), with the options for the model building procedure: `modelL1` Function for fitting the L1 model (default: `buildLM`) which combines the results of the L0 models. `modelL1Control` List of control parameters for the L1 model (default: `list()`). `modelL0` A list of functions for fitting the L0 models (default: `list(buildLM,buildRandomForest,buil` `modelL0Control` List of control lists for each L0 model (default: `list(list(),list(),list())`).

Value

returns an object of class `ensembleStack`.

Note

Loosely based on the code by Emanuele Olivetti https://github.com/emanuele/kaggle_pbr/blob/master/blend.py

References

Bartz-Beielstein, Thomas. Stacked Generalization of Surrogate Models-A Practical Approach. Technical Report 5/2016, TH Koeln, Koeln, 2016.

David H Wolpert. Stacked generalization. Neural Networks, 5(2):241-259, January 1992.

See Also

`predict.ensembleStack`

Examples

```
## Create a test function: branin
braninFunction <- function (x) {
(x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
}
## Create design points
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points
y <- as.matrix(apply(x,1,braninFunction))
## Create model with default settings
fit <- buildEnsembleStack(x,y)
## Predict new point
predict(fit,cbind(1,2))
## True value at location
braninFunction(c(1,2))
```

`buildKriging` *Build Kriging Model*

Description

This function builds a Kriging model based on code by Forrester et al.. By default exponents (p) are fixed at a value of two, and a nugget (or regularization constant) is used. To correct the uncertainty estimates in case of nugget, re-interpolation is also by default turned on.

Usage

```
buildKriging(x, y, control = list())
```

Arguments

`x`	design matrix (sample locations)
`y`	vector of observations at `x`

`control`	(list), with the options for the model building procedure:
	`types` a character vector giving the data type of each variable. All but "factor" will be handled as numeric, "factor" (categorical) variables will be subject to the hamming distance.
	`thetaLower` lower boundary for theta, default is `1e-4`
	`thetaUpper` upper boundary for theta, default is `1e2`
	`algTheta` algorithm used to find theta, default is `optimLBFGSB`.
	`budgetAlgTheta` budget for the above mentioned algorithm, default is `200`. The value will be multiplied with the length of the model parameter vector to be optimized.
	`optimizeP` boolean that specifies whether the exponents (p) should be optimized. Else they will be set to two. Default is `FALSE`

useLambda whether or not to use the regularization constant lambda (nugget effect). Default is TRUE.
lambdaLower lower boundary for log10lambda, default is -6
lambdaUpper upper boundary for log10lambda, default is 0
startTheta optional start value for theta optimization, default is NULL
reinterpolate whether (TRUE,default) or not (FALSE) reinterpolation should be performed target target values of the prediction, a vector of strings. Each string specifies a value to be predicted, e.g., "y" for mean, "s" for standard deviation, "ei" for expected improvement. See also predict.kriging. This can also be changed after the model has been built, by manipulating the respective object$target value.

Details

The model uses a Gaussian kernel: k(x,z)=exp(-sum(theta_i * |x_i-z_i|^p_i)). By default, p_i = 2. Note that if dimension x_i is a factor variable (see parameter types), Hamming distance will be used instead of |x_i-z_i|.

Value

an object of class kriging. Basically a list, with the options and found parameters for the model which has to be passed to the predictor function:
x sample locations (scaled to values between 0 and 1)
y observations at sample locations (see parameters)
thetaLower lower boundary for theta (see parameters)
thetaUpper upper boundary for theta (see parameters)
algTheta algorithm to find theta (see parameters)
budgetAlgTheta budget for the above mentioned algorithm (see parameters)
optimizeP boolean that specifies whether the exponents (p) were optimized (see parameters)
normalizeymin minimum in normalized space
normalizeymax maximum in normalized space
normalizexmin minimum in input space
normalizexmax maximum in input space
dmodeltheta vector of activity parameters
Theta log_10 vector of activity parameters (i.e. log10(dmodeltheta))
dmodellambda regularization constant (nugget)
Lambda log_10 of regularization constant (nugget) (i.e. log10(dmodellambda))
yonemu Ay-ones*mu
ssq sigma square
mu mean mu
Psi matrix large Psi
Psinv inverse of Psi
nevals number of Likelihood evaluations during MLE

References

Forrester, Alexander I.J.; Sobester, Andras; Keane, Andy J. (2008). Engineering Design via Surrogate Modelling - A Practical Guide. John Wiley & Sons.

See Also

predict.kriging

Examples

```
## Test-function:
braninFunction <- function (x) {
(x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
}
## Create design points
set.seed(1)
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points (for Branin function)
y <- as.matrix(apply(x,1,braninFunction))
## Create model with default settings
fit <- buildKriging(x,y,control = list(algTheta=optimLHD))
## Print model parameters
print(fit)
## Predict at new location
predict(fit,cbind(1,2))
## True value at location
braninFunction(c(1,2))
##
## Next Example: Handling factor variables
##
## create a test function:
braninFunctionFactor <- function (x) {
y <- (x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
if(x[3]==1)
y <- y +1
else if(x[3]==2)
y <- y -1
y
}
## create training data
set.seed(1)
x <- cbind(runif(50)*15-5,runif(50)*15,sample(1:3,50,replace=TRUE))
y <- as.matrix(apply(x,1,braninFunctionFactor))
## fit the model (default: assume all variables are numeric)
fitDefault <- buildKriging(x,y,control = list(algTheta=optimLBFGSB))
## fit the model (give information about the factor variable)
fitFactor <- buildKriging(x,y,control =
list(algTheta=optimLBFGSB,types=c("numeric","numeric","factor")))
## create test data
xtest <- cbind(runif(200)*15-5,runif(200)*15,sample(1:3,200,replace=TRUE))
ytest <- as.matrix(apply(xtest,1,braninFunctionFactor))
## Predict test data with both models, and compute error
ypredDef <- predict(fitDefault,xtest)$y
ypredFact <- predict(fitFactor,xtest)$y
mean((ypredDef-ytest)^2)
```

```
mean((ypredFact-ytest)^2)
```

`buildKrigingDACE` *Build DACE model*

Description

This Kriging meta model is based on DACE (Design and Analysis of Computer Experiments). It allows to choose different regression and correlation models. The optimization of model parameters is by default done with a bounded simplex method from the `nloptr` package.

Usage

```
buildKrigingDACE(x, y, control = list())
```

Arguments

`x`	design matrix (sample locations), rows for each sample, columns for each variable.
`y`	vector of observations at `x`
`control`	(list), with the options for the model building procedure: `startTheta` optional start value for theta optimization, default is `NULL` `algTheta` algorithm used to find theta, default is `optimLBFGSB`. `budgetAlgTheta` budget for the above mentioned algorithm, default is `200`. The value will be multiplied with the length of the model parameter vector to be optimized. `nugget` Value for nugget. Default is -1, which means the nugget will be optimized during MLE. Else it can be fixed in a range between 0 and 1. `regr` Regression function to be used: `regpoly0` (default), `regpoly1`, `regpoly2`. Can be a custom user function. `corr` Correlation function to be used: `corrnoisykriging` (default), `corrkriging`, `corrnoisygauss`, `corrgauss`, `correxp`, `correxpg`, `corrlin`, `corrcubic`,`corrspherical`,`corrspline`. Can also be user supplied (if in the right form). `target` target values of the prediction, a vector of strings. Each string specifies a value to be predicted, e.g., "y" for mean, "s" for standard deviation, "ei" for expected improvement. See also `predict.kriging`. This can also be changed after the model has been build, by manipulating the respective `object$target` value.

Value

returns an object of class `dace` with the following elements:

`model`	A list, containing model parameters
`like`	Estimated likelihood value
`theta`	activity parameters theta (vector)
`p`	exponents p (vector)
`lambda`	nugget value (numeric)
`nevals`	Number of iterations during MLE

Author(s)

The authors of the original DACE Matlab toolbox are Hans Bruun Nielsen <hbn@imm.dtu.dk>, Soren Nymand Lophaven and Jacob Sondergaard.
Extension of the Matlab code by Tobias Wagner <wagner@isf.de>.
Porting and adaptation to R and further extensions by Martin Zaefferer <martin.zaefferer@fh-koeln.de>.

References

S.~Lophaven, H.~Nielsen, and J.~Sondergaard. DACE—A Matlab Kriging Toolbox. Technical Report IMM-REP-2002-12, Informatics and Mathematical Modelling, Technical University of Denmark, Copenhagen, Denmark, 2002.

See Also

predict.dace

Examples

```
## Create design points
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points
y <- funSphere(x)
## Create model with default settings
fit <- buildKrigingDACE(x,y)
## Print model parameters
print(fit)
## Create with different regression and correlation functions
fit <- buildKrigingDACE(x,y,control=list(regr=regpoly2,corr=corrspline))
## Print model parameters
print(fit)
```

---

buildLM *Linear Model Interface*

Description

This is a simple wrapper for the lm function, which fits linear models. The purpose of this function is to provide an interface as required by SPOT, to enable modeling and model-based optimization with linear models. The linear model is build with main effects. Optionally, the model is also subject to the AIC-based stepwise algorithm, using the `step` function from the `stats` package.

Usage

```
buildLM(x, y, control = list())
```

Arguments

x	matrix of input parameters. Rows for each point, columns for each parameter.
y	one column matrix of observations to be modeled.
control	list of control parameters, currently only with parameters useStep and formula. The useStep boolean specifies whether the step function is used. The formula is passed to the lm function. Without a formula, a second order model will be built.

Value

an object of class "spotLinearModel", with a predict method and a print method.

Examples

```
## Test-function:
braninFunction <- function (x) {
(x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
}
## Create design points
set.seed(1)
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points (for Branin function)
y <- as.matrix(apply(x,1,braninFunction))
## Create model
fit <- buildLM(x,y,control = list(algTheta=optimLHD))
## Print model parameters
print(fit)
## Predict at new location
predict(fit,cbind(1,2))
## True value at location
braninFunction(c(1,2))
```

---

buildLOESS *Build LOESS Model*

Description

Build an interpolation model using the loess function. Essentially a SPOT-style interface to that function.

Usage

```
buildLOESS(x, y, control = list())
```

Arguments

x	design matrix (sample locations), rows for each sample, columns for each variable.
y	vector of observations at x
control	named list, with the options for the model building procedure loess. These will be passed to loess as arguments. Please refrain from setting the formula or data arguments as these will be supplied by the interface, based on x and y.

Value

returns an object of class spotLOESS.

See Also

predict.spotLOESS

Examples

```
## Create a test function: branin
braninFunction <- function (x) {
(x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
}
## Create design points
set.seed(1)
x <- cbind(runif(40)*15-5,runif(40)*15)
## Compute observations at design points
y <- as.matrix(apply(x,1,braninFunction))
## Create model with default settings
fit <- buildLOESS(x,y)
fit
## Predict new point
predict(fit,cbind(1,2))
## True value at location
braninFunction(c(1,2))
## Change model control
fit <- buildLOESS(x,y,control=list(parametric=c(TRUE,FALSE)))
fit
```

---

buildRandomForest *Random Forest Interface*

Description

This is a simple wrapper for the randomForest function from the randomForest package. The purpose of this function is to provide an interface as required by SPOT, to enable modeling and model-based optimization with random forest.

Usage

```
buildRandomForest(x, y, control = list())
```

Arguments

x	matrix of input parameters. Rows for each point, columns for each parameter.
y	one column matrix of observations to be modeled.
control	list of control parameters, currently not used.

Value

an object of class "spotRandomForest", with a predict method and a print method.

Examples

```
## Not run:
## Test-function:
braninFunction <- function (x) {
(x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
}
## Create design points
set.seed(1)
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points (for Branin function)
y <- as.matrix(apply(x,1,braninFunction))
## Create model
fit <- buildRandomForest(x,y,control = list(algTheta=optimLHD))
## Print model parameters
print(fit)
## Predict at new location
predict(fit,cbind(1,2))
## True value at location
braninFunction(c(1,2))

## End(Not run)
```

---

buildRSM *Build Response Surface Model*

Description

Using the rsm package, this function builds a linear response surface model.

Usage

```
buildRSM(x, y, control = list())
```

Arguments

x	design matrix (sample locations), rows for each sample, columns for each variable.
y	vector of observations at x
control	(list), with the options for the model building procedure:
	mainEffectsOnly Logical, defaults to FALSE. Set to TRUE if a model with main effects only is desired (no interactions, second order effects).
	canonical Logical, defaults to FALSE. If this is TRUE, use the canonical path to descent from saddle points. Else, simply use steepest descent

Value

returns an object of class spotRSM.

See Also

predict.spotRSM

Examples

```
## Create a test function: branin
braninFunction <- function (x) {
(x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
}
## Create design points
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points
y <- as.matrix(apply(x,1,braninFunction))
## Create model with default settings
fit <- buildRSM(x,y)
## Predict new point
predict(fit,cbind(1,2))
## True value at location
braninFunction(c(1,2))
## plots
plot(fit)
## path of steepest descent
descentSpotRSM(fit)
```

---

dataGasSensor *Gas Sensor Data*

Description

A data set of a Gas Sensor, similar to the one used by Rebolledo et al. 2016. It also contains information of 10 different test/training splits, to enable comparable evaluation procedures.

Usage

```
dataGasSensor
```

Format

A data frame with 280 rows and 20 columns (1 output, 7 input, 2 disturbance, 10 training/test split) :

Y Measured Sensor Output

X1 Sensor Input 1

X2 Sensor Input 2

X3 Sensor Input 3

X4 Sensor Input 4

X5 Sensor Input 5

X6 Sensor Input 6

X7 Sensor Input 7

Batch Disturbance variable, measurement batch

Sensor Disturbance variable, sensor ID

Set1 test/training split, 1 is training data, 2 is test data

Set2 test/training split

Set3 test/training split

Set4 test/training split

Set5 test/training split

Set6 test/training split

Set7 test/training split

Set8 test/training split

Set9 test/training split

Set10 test/training split

Two different modeling tasks are of interest for this data set: `Y~X1+X2+X3+X4+X5+X6+X7+Batch+Sensor` and `X1~Y+X7+Batch+Sensor`.

References

Margarita A. Rebolledo C., Sebastian Krey, Thomas Bartz-Beielstein, Oliver Flasch, Andreas Fischbach and Joerg Stork.
2016.
Modeling and Optimization of a Robust Gas Sensor.
7th International Conference on Bioinspired Optimization Methods and their Applications (BIOMA 2016).

descentSpotRSM *Descent RSM model*

Description

Generate steps along the path of steepest descent for a RSM model. This is only intended as a manual tool to use together with buildRSM.

Usage

```
descentSpotRSM(object)
```

Arguments

object	RSM model (settings and parameters) of class spotRSM.

Value

list with

x list of points along the path of steepest descent

y corresponding predicted values

See Also

buildRSM

designLHD *Latin Hypercube Design Generator*

Description

Creates a latin Hypercube Design (LHD) with user-specified dimension and number of design points. LHDs are created repeatedly created at random. For each each LHD, the minimal pairwise distance between design points is computed. The design with the maximum of that minimal value is chosen.

Usage

```
designLHD(x = NULL, lower, upper, control = list())
```

Arguments

x	optional matrix x, rows for points, columns for dimensions. This can contain one or more points which are part of the design, but specified by the user. These points are added to the design, and are taken into account when calculating the pair-wise distances. They do not count for the design size. E.g., if x has two rows, control$replicates is one and control$size is ten, the returned design will have 12 points (12 rows). The first two rows will be identical to x. Only the remaining ten rows are guaranteed to be a valid LHD.
lower	vector with lower boundary of the design variables (in case of categorical parameters, please map the respective factor to a set of contiguous integers, e.g., with lower = 1 and upper = number of levels)
upper	vector with upper boundary of the design variables (in case of categorical parameters, please map the respective factor to a set of contiguous integers, e.g., with lower = 1 and upper = number of levels)
control	list of controls: size number of design points retries number of retries during design creation types this specifies the data type for each design parameter, as a vector of either "numeric","integer","factor". (here, this only affects rounding) inequalityConstraint inequality constraint function, smaller zero for infeasible points. Used to replace infeasible points with random points. replicates integer for replications of each design point. E.g., if replications is two, every design point will occur twice in the resulting matrix.

Value

matrix design
- design has length(lower) columns and (size + nrow(x))*control$replicates rows. All values should be within lower <= design <= upper

Author(s)

Original code by Christian Lasarczyk, adaptations by Martin Zaefferer

Examples

```
set.seed(1) #set RNG seed to make examples reproducible
design <- designLHD(,1,2) #simple, 1-D case
design
design <- designLHD(,1,2,control=list(replicates=3)) #with replications
design
design <- designLHD(,c(-1,-2,1,0),c(1,4,9,1),
control=list(size=5, retries=100, types=c("numeric","integer","factor","factor")))
design
x <- designLHD(,c(1,-10),c(2,10),control=list(size=5,retries=100))
x2 <- designLHD(x,c(1,-10),c(2,10),control=list(size=5,retries=100))
plot(x2)
points(x, pch=19)
```

designUniformRandom *Uniform Design Generator*

Description

Create a simple experimental design based on uniform random sampling.

Usage

```
designUniformRandom(x = NULL, lower, upper, control = list())
```

Arguments

x	optional data.frame x to be part of the design
lower	vector with lower boundary of the design variables (in case of categorical parameters, please map the respective factor to a set of contiguous integers, e.g., with lower = 1 and upper = number of levels)
upper	vector with upper boundary of the design variables (in case of categorical parameters, please map the respective factor to a set of contiguous integers, e.g., with lower = 1 and upper = number of levels)
control	list of controls: size number of design points types this specifies the data type for each design parameter, as a vector of either "numeric","integer","factor". (here, this only affects rounding) replicates integer for replications of each design point. E.g., if replications is two, every design point will occur twice in the resulting matrix.

Value

matrix design
- design has length(lower) columns and (size + nrow(x))*control$replicates rows. All values should be within lower <= design <= upper

Examples

```
set.seed(1) #set RNG seed to make examples reproducible
design <- designUniformRandom(,1,2) #simple, 1-D case
design
design <- designUniformRandom(,1,2,control=list(replicates=3)) #with replications
design
design <- designUniformRandom(,c(-1,-2,1,0),c(1,4,9,1),
control=list(size=5, types=c("numeric","integer","factor","factor")))
design
x <- designUniformRandom(,c(1,-10),c(2,10),control=list(size=5))
x2 <- designUniformRandom(x,c(1,-10),c(2,10),control=list(size=5))
plot(x2)
points(x, pch=19)
```

expectedImprovement *Expected Improvement*

Description

Compute the negative logarithm of the Expected Improvement of a set of candidate solutions. Based on mean and standard deviation of a candidate solution, this estimates the expectation of improvement. Improvement considers the amount by which the best known value (best observed value) is exceeded by the candidates.

Usage

```
expectedImprovement(mean, sd, min)
```

Arguments

mean	vector of predicted means of the candidate solutions.
sd	vector of estimated uncertainties / standard deviations of the candidate solutions.
min	minimal observed value.

Value

a vector with the negative logarithm of the expected improvement values, -log10(EI).

Examples

```
mean <- 1:10 #mean of the candidates
sd <- 10:1 #st. deviation of the candidates
min <- 5 #best known value
EI <- expectedImprovement(mean,sd,min)
EI
```

funCyclone *Objective function - Cyclone Simulation: Barth/Muschelknautz*

Description

Calculate cyclone collection efficiency. A simple, physics-based optimization problem (potentially bi-objective). See the references [1,2].

Usage

```
funCyclone(x, deterministic = c(T, T, T), cyclone = list(Da = 1.26, H =
  2.5, Dt = 0.42, Ht = 0.65, He = 0.6, Be = 0.2), fluid = list(Mu =
  1.85e-05, Ve = (50/36)/0.12, lambdag = 1/200, Rhop = 2000, Rhof = 1.2,
  Croh = 0.05), noiseLevel = list(Vp = 0.1, Rhop = 0.05),
  model = "Barth-Muschelknautz", intervals = c(0, 2, 4, 6, 8, 10, 15,
  20, 30) * 1e-06, delta = c(0, 0.02, 0.03, 0.05, 0.1, 0.3, 0.3, 0.2))
```

Arguments

x	vector of length at least one and up to six, specifying non-default geometrical parameters in [m]: Da, H, Dt, Ht, He, Be
deterministic	binary vector. First element specifies whether volume flow is deterministic or not. Second element specifies whether particle density is deterministic or not. Third element specifies whether particle diameters are deterministic or not. Default: All are deterministic (TRUE).
cyclone	list of a default cyclone's geometrical parameters: fluid$Da, fluid$H, fluid$Dt, fluid$Ht, fluid$He and fluid$Be
fluid	list of default fluid parameters: fluid$Mu, fluid$Vp, fluid$Rhop, fluid$Rhof and fluid$Croh
noiseLevel	list of noise levels for volume flow (noiseLevel$Vp) and particle density (noiseLevel$Rhop), only used if non-deterministic.
model	type of the model (collection efficiency only): either "Barth-Muschelknautz" or "Mothes"
intervals	vector specifying the particle size interval bounds.
delta	vector of densities in each interval (specified by intervals). Should have one element less than the intervals parameter.

Value

returns a function that calculates the fractional efficiency for the specified diameter, see example.

References

[1] Zaefferer, M.; Breiderhoff, B.; Naujoks, B.; Friese, M.; Stork, J.; Fischbach, A.; Flasch, O.; Bartz-Beielstein, T. Tuning Multi-objective Optimization Algorithms for Cyclone Dust Separators Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, ACM, 2014, 1223-1230

[2] Breiderhoff, B.; Bartz-Beielstein, T.; Naujoks, B.; Zaefferer, M.; Fischbach, A.; Flasch, O.; Friese, M.; Mersmann, O.; Stork, J.; Simulation and Optimization of Cyclone Dust Separators Proceedings 23. Workshop Computational Intelligence, 2013, 177-196

Examples

```
## Call directly
funCyclone(c(1.26,2.5))
## create vectorized target funcion, vectorized, first objective only
## Also: negated, since SPOT always does minimization.
tfunvecF1 <-function(x){-apply(x,1,funCyclone)[1,]}
tfunvecF1(matrix(c(1.26,2.5,1,2),2,2,byrow=TRUE))
## optimize with spot
res <- spot(fun=tfunvecF1,lower=c(1,2),upper=c(2,3),
   control=list(modelControl=list(target="ei"),
   model=buildKriging,optimizer=optimLBFGSB,plots=TRUE))
## best found solution ...
```

```
res$xbest
## ... and its objective function value
res$ybest
```

---

funSphere *Sphere Test Function*

Description

Sphere Test Function

Usage

```
funSphere(x)
```

Arguments

x	matrix of points to evaluate with the sphere function. Rows for points and columns for dimension.

Value

1-column matrix with resulting function values

Examples

```
funSphere(matrix(runif(18),,3))
```

---

optimDE *Minimization by Differential Evolution*

Description

For minimization, this function uses the "DEoptim" method from the codeDEoptim package. It is basically a wrapper, to enable DEoptim for usage in SPOT.

Usage

```
optimDE(x = NULL, fun, lower, upper, control = list(), ...)
```

Arguments

x	optional start point, not used in DEoptim
fun	objective function, which receives a matrix x and returns observations y
lower	boundary of the search space
upper	boundary of the search space
control	list of control parameters
	funEvals Budget, number of function evaluations allowed. Default is 200.
	populationSize Population size or number of particles in the population. Default is 10*dimension.
...	passed to fun

Value

list, with elements

x archive of the best member at each iteration

y archive of the best value of fn at each iteration

xbest best solution

ybest best observation

count number of evaluations of fun

Examples

```
res <- optimDE(,lower = c(-10,-20),upper=c(20,8),fun = funSphere)
res$ybest
```

---

optimES *Evolution Strategy*

Description

This is an implementation of an Evolution Strategy.

Usage

```
optimES(x = NULL, fun, lower, upper, control = list(), ...)
```

Arguments

x	optional start point, not used
fun	objective function, which receives a matrix x and returns observations y
lower	is a vector that defines the lower boundary of search space (this also defines the dimensionality of the problem)
upper	is a vector that defines the upper boundary of search space (same length as lower)
control	list of control parameters. The control list can contain the following settings:

funEvals number of function evaluations, stopping criterion, default is 500

mue number of parents, default is 10

nu selection pressure. That means, number of offspring (lambda) is mue multiplied with nu. Default is 10

mutation string of mutation type, default is 1

sigmaInit initial sigma value (step size), default is 1.0

nSigma number of different sigmas, default is 1

tau0 number, default is 0.0. tau0 is the general multiplier.

tau number, learning parameter for self adaption, i.e. the local multiplier for step sizes (for each dimension).default is 1.0

rho number of parents involved in the procreation of an offspring (mixing number), default is "bi"

sel number of selected individuals, default is 1

stratReco Recombination operator for strategy variables. 1: none. 2: dominant/discrete (default). 3: intermediate. 4: variation of intermediate recombination.

objReco Recombination operator for object variables. 1: none. 2: dominant/discrete (default). 3: intermediate. 4: variation of intermediate recombination.

maxGen number of generations, stopping criterion, default is Inf

seed number, random seed, default is 1

noise number, value of noise added to fitness values, default is 0.0

verbosity defines output verbosity of the ES, default is 0

plotResult boolean, specifies if results are plotted, default is FALSE

logPlotResult boolean, defines if plot results should be logarithmic, default is FALSE

sigmaRestart number, value of sigma on restart, default is 0.1

preScanMult initial population size is multiplied by this number for a pre-scan, default is 1

globalOpt termination criterion on reaching a desired optimum value, default is rep(0,dimension)

...	additional parameters to be passed on to fun

Value

list, with elements

x NULL, currently not used

y NULL, currently not used

xbest best solution

ybest best observation

count number of evaluations of fun

Examples

```
cont <- list(funEvals=100)
optimES(fun=funSphere,lower=rep(0,2), upper=rep(1,2), control= cont)
```

---

optimGenoud *Minimization by GENetic Optimization Using Derivatives*

Description

For minimization, this function uses the "genoud" method from the codergenoud package. It is basically a wrapper, to enable genoud for usage in SPOT.

Usage

```
optimGenoud(x = NULL, fun, lower, upper, control = list(), ...)
```

Arguments

x	optional start point, not used
fun	objective function, which receives a matrix x and returns observations y
lower	boundary of the search space
upper	boundary of the search space
control	list of control parameters
	funEvals Budget, number of function evaluations allowed. Default is 100.
	populationSize Population size, number of individuals in the population. Default is 10*dimension.
...	passed to fun

Value

list, with elements

x NULL, currently not used

y NULL, currently not used

xbest best solution

ybest best observation

count number of evaluations of fun

Examples

```
res <- optimGenoud(,fun = funSphere,lower = c(-10,-20),upper=c(20,8))
res$ybest
```

---

optimLBFGSB *Minimization by L-BFGS-B*

Description

For minimization, this function uses the "L-BFGS-B" method from the optim function, which is part of the codestats package. It is basically a wrapper, to enable L-BFGS-B for usage in SPOT.

Usage

```
optimLBFGSB(x = NULL, fun, lower, upper, control = list(), ...)
```

Arguments

x	optional matrix of points. Only first point (row) is used as startpoint.
fun	objective function, which receives a matrix x and returns observations y
lower	boundary of the search space
upper	boundary of the search space
control	list of control parameters
	funEvals Budget, number of function evaluations allowed. Default is 100.
	All other control parameters accepted by the optim function can be used, too, and are passed to optim.
...	passed to fun

Value

list, with elements

- `x` NA, not used
- `y` NA, not used
- `xbest` best solution
- `ybest` best observation
- `count` number of evaluations of `fun` (estimated from the more complicated `"counts"` variable returned by `optim`)
- `message` termination message returned by `optim`

Examples

```
res <- optimLBFGSB(,fun = funSphere,lower = c(-10,-20),upper=c(20,8))
res$ybest
```

`optimLHD` *Minimization by Latin Hypercube Sampling*

Description

This uses Latin Hypercube Sampling (LHS) to optimize a specified target function. A Latin Hypercube Design (LHD) is created with `designLHD`, then evaluated by the objective function. All results are reported, including the best (minimal) objective value, and corresponding design point.

Usage

```
optimLHD(x = NULL, fun, lower, upper, control = list(), ...)
```

Arguments

`x`	optional matrix of points to be included in the evaluation
`fun`	objective function, which receives a matrix x and returns observations y
`lower`	boundary of the search space
`upper`	boundary of the search space
`control`	list of control parameters
	`funEvals` Budget, number of function evaluations allowed. Default: 100.
	`retries` Number of retries for design generation, used by `designLHD`. Default: 100.
`...`	passed to `fun`

Value

list, with elements

`x` archive of evaluated solutions

`y` archive of observations

`xbest` best solution

`ybest` best observation

`count` number of evaluations of `fun`

`message` success message

Examples

```
res <- optimLHD(,fun = funSphere,lower = c(-10,-20),upper=c(20,8))
res$ybest
```

`optimNLOPTR` *Minimization by NLOPT*

Description

This is a wrapper that employs the `nloptr` function from the package of the same name. The `nloptr` function itself is an interface to the `nlopt` library, which contains a wide selection of different optimization algorithms.

Usage

```
optimNLOPTR(x = NULL, fun, lower, upper, control = list(), ...)
```

Arguments

`x`	optional matrix of points to be included in the evaluation (only first row will be used)
`fun`	objective function, which receives a matrix x and returns observations y
`lower`	boundary of the search space
`upper`	boundary of the search space
`control`	named list, with the options for `nloptr`. These will be passed to `nloptr` as arguments. In addition, the following parameter can be used to set the function evaluation budget: `funEvals` Budget, number of function evaluations allowed. Default: 100.
`...`	passed to `fun` Note that the arguments `x`, `fun`, `lower` and `upper` will be mapped to the corresponding arguments of `nloptr`: `x0`, `eval_f`, `lb` and `ub`.

Value

list, with elements

`x` archive of evaluated solutions

`y` archive of observations

`xbest` best solution

`ybest` best observation

`count` number of evaluations of `fun`

`message` success message

Examples

```
## Not run:
##simple example:
res <- optimNLOPTR(,fun = funSphere,lower = c(-10,-20),upper=c(20,8))
res
##with an inequality constraint:
contr <- list()  #control list
##specify constraint
contr$eval_g_ineq <- function(x) 1+x[1]-x[2]
res <- optimNLOPTR(,fun=funSphere,lower=c(-10,-20),upper=c(20,8),control=contr)
res

## End(Not run)
```

---

plotData *Interpolated plot*

Description

A (filled) contour or perspective plot of a data set with two independent and one dependent variable. The plot is generated by some interpolation or regression model. By default, the `loess` function is used.

Usage

```
plotData(x, y, which = 1:2, constant = x[which.min(y), ],
  model = buildLOESS, modelControl = list(), xlab = c("x1", "x2"),
  ylab = "y", type = "filled.contour", ...)
```

Arguments

`x`	independent variables, or input variables. this should be a matrix of at least two columns and several rows. If more than two columns are present, all will be used for fitting the model. The parameter `which` will determine which of these will be plotted, and the parameter `constant` will determine the values of all parameters that are not varied.

y	dependent, or observed output variable to be interpolated/regressed and plotted.
which	a vector with two elements, each an integer giving the two independent variables of the plot (the integers are indices of the respective data set, i.e., columns of x). All other parameters will be fixed to the best known solution, i.e., the one with minimal y-value.
constant	a numeric vector that states for each variable a constant value that it will take on if it is not varied in the plot. This affects the parameters not selected by the `which` parameter. By default, this will be fixed to the best known solution, i.e., the one with minimal y-value, according to `which.min(object$y)`. The length of this numeric vector should be the same as the number of columns in `object$x`
model	the model building function to be used, by default `buildLOESS`.
modelControl	control list of the chosen model building function.
xlab	a vector of characters, giving the labels for each of the two independent variables
ylab	character, the value of the dependent variable predicted by the corresponding model
type	string describing the type of the plot: `"filled.contour"` (default), `"contour"`, `"persp"` (perspective), or `"persp3d"` plot. Note that "persp3d" is based on the plotly package and will work in RStudio, but not in the standard RGui.
...	additional parameters passed to the `contour` or `filled.contour` function

See Also

plotFunction, plotModel

Examples

```
## generate random test data
testfun <- function (x) sum(x^2)
set.seed(1)
k <- 30
x <- cbind(runif(k)*15-5,runif(k)*15)
y <- as.matrix(apply(x,1,testfun))
plotData(x,y)
plotData(x,y,type="contour")
plotData(x,y,type="persp")
```

---

plotFunction *Surface plot of a function*

Description

A (filled) contour plot or perspective / surface plot of a function.

Usage

```
plotFunction(f = function(x) {     rowSums(x^2) }, lower = c(0, 0),
  upper = c(1, 1), type = "filled.contour", s = 100, xlab = "x1",
  ylab = "x2", zlab = "y", color.palette = terrain.colors,
  title = " ", levels = NULL, points1, points2, pch1 = 20,
  pch2 = 8, lwd1 = 1, lwd2 = 1, cex1 = 1, cex2 = 1,
  col1 = "red", col2 = "black", theta = -40, phi = 40, ...)
```

Arguments

f	function to be plotted. The function should either be able to take two vectors or one matrix specifying sample locations. i.e. z=f(X) or z=f(x2,x1) where Z is a two column matrix containing the sample locations x1 and x2.
lower	boundary for x1 and x2 (defaults to c(0,0)).
upper	boundary (defaults to c(1,1)).
type	string describing the type of the plot: "filled.contour" (default), "contour", "persp" (perspective), or "persp3d" plot. Note that "persp3d" is based on the plotly package and will work in RStudio, but not in the standard RGui.
s	number of samples along each dimension. e.g. f will be evaluated s^2 times.
xlab	lable of first axis
ylab	lable of second axis
zlab	lable of third axis
color.palette	colors used, default is terrain.color
title	of the plot
levels	number of levels for the plotted function value. Will be set automatically with default NULL.. (contour plots only)
points1	can be omitted, but if given the points in this matrix are added to the plot in form of dots. Contour plots and persp3d only. Contour plots expect matrix with two columns for coordinates. 3Dperspective expects matrix with three columns, third column giving the corresponding observed value of the plotted function.
points2	can be omitted, but if given the points in this matrix are added to the plot in form of crosses. Contour plots and persp3d only. Contour plots expect matrix with two columns for coordinates. 3Dperspective expects matrix with three columns, third column giving the corresponding observed value of the plotted function.
pch1	pch (symbol) setting for points1 (default: 20). (contour plots only)
pch2	pch (symbol) setting for points2 (default: 8). (contour plots only)
lwd1	line width for points1 (default: 1). (contour plots only)
lwd2	line width for points2 (default: 1). (contour plots only)
cex1	cex for points1 (default: 1). (contour plots only)
cex2	cex for points2 (default: 1). (contour plots only)
col1	color for points1 (default: "black"). (contour plots only)
col2	color for points2 (default: "black"). (contour plots only)

theta	angle defining the viewing direction. theta gives the azimuthal direction and phi the colatitude. (persp plot only)
phi	angle defining the viewing direction. theta gives the colatitude. (persp plot only)
...	additional parameters passed to contour or filled.contour

See Also

plotData, plotModel

Examples

```
plotFunction(function(x){rowSums(x^2)},c(-5,0),c(10,15))
plotFunction(function(x){rowSums(x^2)},c(-5,0),c(10,15),type="contour")
plotFunction(function(x){rowSums(x^2)},c(-5,0),c(10,15),type="persp")
```

---

plotModel *Surface plot of a model*

Description

A (filled) contour or perspective plot of a fitted model.

Usage

```
plotModel(object, which = 1:2, constant = object$x[which.min(object$y),
  ], xlab = paste("x", which, sep = ""), ylab = "y",
  type = "filled.contour", ...)
```

Arguments

object	fit created by a modeling function, e.g., buildRandomForest.
which	a vector with two elements, each an integer giving the two independent variables of the plot (the integers are indices of the respective data set).
constant	a numeric vector that states for each variable a constant value that it will take on if it is not varied in the plot. This affects the parameters not selected by the which parameter. By default, this will be fixed to the best known solution, i.e., the one with minimal y-value, according to which.min(object$y). The length of this numeric vector should be the same as the number of columns in object$x
xlab	a vector of characters, giving the labels for each of the two independent variables.
ylab	character, the value of the dependent variable predicted by the corresponding model.
type	string describing the type of the plot: "filled.contour" (default), "contour", "persp" (perspective), or "persp3d" plot. Note that "persp3d" is based on the plotly package and will work in RStudio, but not in the standard RGui.
...	additional parameters passed to the contour or filled.contour function.

See Also

plotFunction, plotData

Examples

```
## generate random test data
testfun <- function (x) sum(x^2)
set.seed(1)
k <- 30
x <- cbind(runif(k)*15-5,runif(k)*15,runif(k)*2-7,runif(k)*5+22)
y <- as.matrix(apply(x,1,testfun))
fit <- buildLM(x,y)
plotModel(fit)
plotModel(fit,type="contour")
plotModel(fit,type="persp")
plotModel(fit,which=c(1,4))
plotModel(fit,which=2:3)
```

---

repeatsOCBA *Optimal Computing Budget Allocation*

Description

A simple interface to the Optimal Computing Budget Allocation algorithm.

Usage

```
repeatsOCBA(x, y, budget)
```

Arguments

x	matrix of samples. Identical rows indicate repeated evaluations. Any sample should be evaluated at least twice, to get an estimate of the variance.
y	observations of the respective samples. For repeated evaluations, y should differ (variance not zero).
budget	of additional evaluations to be allocated to the samples.

Value

A vector that specifies how often each solution should be evaluated.

References

Chun-hung Chen and Loo Hay Lee. 2010. Stochastic Simulation Optimization: An Optimal Computing Budget Allocation (1st ed.). World Scientific Publishing Co., Inc., River Edge, NJ, USA.

See Also

repeatsOCBA calls OCBA, which also provides some additional details.

Examples

```
x <- matrix(c(1:3,1:3),9,2)
y <- runif(9)
repeatsOCBA(x,y,10)
```

---

satter *Satterthwaite Function*

Description

The Satterthwaite function can be used to estimate the magnitude of the variance component (sigma_beta)^2, when the random factor has significant main effects.

Usage

```
satter(MScoeff, MSi, dfi, alpha = 0.05)
```

Arguments

MScoeff	coefficients c_1, c_2
MSi	mean squared values
dfi	degrees of freedom
alpha	error probability

Details

Note, the output from the satter() procedure is sigma_beta.

Value

vector with 1. estimate of variance 2. degrees of freedom, 3. lower value of 1-alpha confint 4. upper value of 1-alpha confint

Examples

```
res <- satter(MScoeff= c(1/4, -1/4)
              , MSi = c(394.9, 73.3)
              , dfi = c(4,3)
              , alpha = 0.1)
```

spot *Sequential Parameter Optimization*

Description

This is one of the main interfaces for using the SPOT package. Based on a user-given objective function and configuration, spot finds the parameter setting that yields the lowest objective value (minimization). To that end, it uses methods from the fields of design of experiment, statistical modeling / machine learning and optimization.

Usage

```
spot(x = NULL, fun, lower, upper, control = list(), ...)
```

Arguments

x	is an optional start point (or set of start points), specified as a matrix. One row for each point, and one column for each optimized parameter.
fun	is the objective function. It should receive a matrix x and return a matrix y. In case the function uses external code and is noisy, an additional seed parameter may be used, see the control$seedFun argument below for details.
lower	is a vector that defines the lower boundary of search space. This determines also the dimensionality of the problem.
upper	is a vector that defines the upper boundary of search space.
control	is a list with control settings for spot. See spotControl.
...	additional parameters passed to fun.

Value

This function returns a list with:

xbest Parameters of the best found solution (matrix).

ybest Objective function value of the best found solution (matrix).

x Archive of all evaluation parameters (matrix).

y Archive of the respective objective function values (matrix).

count Number of performed objective function evaluations.

msg Message specifying the reason of termination.

modelFit The fit of the last build model, i.e., an object returned by the last call to the function specified by control$model.

Examples

```
## Most simple example: Kriging + LHS + predicted
## mean optimization (not expected improvement)
res <- spot(,funSphere,c(-2,-3),c(1,2))
res$xbest
## With expected improvement
res <- spot(,funSphere,c(-2,-3),c(1,2),
   control=list(modelControl=list(target="ei")))
res$xbest
### With additional start point:
#res <- spot(matrix(c(0.05,0.1),1,2),funSphere,c(-2,-3),c(1,2))
#res$xbest
#res <- spot(,funSphere,c(-2,-3),c(1,2),
#    control=list(funEvals=50))
#res$xbest
### Use a local optimizer instead of LHS
#res <- spot(,funSphere,c(-2,-3),c(1,2),
#    control=list(optimizer=optimLBFGSB))
#res$xbest
### Random Forest instead of Kriging
#res <- spot(,funSphere,c(-2,-3),c(1,2),
#    control=list(model=buildRandomForest))
#res$xbest
### LM instead of Kriging
#res <- spot(,funSphere,c(-2,-3),c(1,2),
#    control=list(model=buildLM)) #lm as surrogate
#res$xbest
### LM and local optimizer (which for this simple example is perfect)
#res <- spot(,funSphere,c(-2,-3),c(1,2),
#    control=list(model=buildLM, optimizer=optimLBFGSB))
#res$xbest
### Or a different Kriging model:
#res <- spot(,funSphere,c(-2,-3),c(1,2),
#    control=list(model=buildKrigingDACE, optimizer=optimLBFGSB))
#res$xbest
## With noise: (this takes some time)
#res1 <- spot(,function(x)funSphere(x)+rnorm(nrow(x)),c(-2,-3),c(1,2),
# control=list(funEvals=100,noise=TRUE)) #noisy objective
#res2 <- spot(,function(x)funSphere(x)+rnorm(nrow(x)),c(-2,-3),c(1,2),
# control=list(funEvals=100,noise=TRUE,replicates=2,
# designControl=list(replicates=2))) #noise with replicated evaluations
#res3 <- spot(,function(x)funSphere(x)+rnorm(nrow(x)),c(-2,-3),c(1,2),
# control=list(funEvals=100,noise=TRUE,replicates=2,OCBA=T,OCBABudget=1,
# designControl=list(replicates=2))) #and with OCBA
### Check results with non-noisy function:
#funSphere(res1$xbest)
#funSphere(res2$xbest)
#funSphere(res3$xbest)
## The following is for demonstration only, to be used for random number
## seed handling in case of external noisy target functions.
#res3 <- spot(,function(x,seed){set.seed(seed);funSphere(x)+rnorm(nrow(x))},
#    c(-2,-3),c(1,2),control=list(funEvals=100,noise=TRUE,seedFun=1))
```

```
##
## Next Example: Handling factor variables
## Note: factors should be coded as integer values, i.e., 1,2,...,n
## create a test function:
braninFunctionFactor <- function (x) {
  y <- (x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
    10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
  if(x[3]==1)
    y <- y +1
  else if(x[3]==2)
    y <- y -1
  y
}
## vectorize
objFun <- function(x){apply(x,1,braninFunctionFactor)}
set.seed(1)
res <- spot(fun=objFun,lower=c(-5,0,1),upper=c(10,15,3),
    control=list(model=buildKriging,
      types= c("numeric","numeric","factor"),
      optimizer=optimLHD))
res$xbest
res$ybest
```

---

spotAlgEs *Evolution Strategy Implementation*

Description

This function is used by optimES as a main loop for running the Evolution Strategy with the given parameter set specified by SPOT.

Usage

```
spotAlgEs(mue = 10, nu = 10, dimension = 2, mutation = 2,
  sigmaInit = 1, nSigma = 1, tau0 = 0, tau = 1, rho = "bi",
  sel = -1, stratReco = 1, objReco = 2, maxGen = Inf,
  maxIter = Inf, seed = 1, noise = 0, fName = funSphere,
  lowerLimit = -1, upperLimit = 1, verbosity = 0,
  plotResult = FALSE, logPlotResult = FALSE, sigmaRestart = 0.1,
  preScanMult = 1, globalOpt = NULL, ...)
```

Arguments

mue	number of parents, default is 10
nu	selection pressure. That means, number of offspring (lambda) is mue multiplied with nu. Default is 10
dimension	dimension number of the target function, default is 2
mutation	mutation type, either 1 or 2, default is 1

sigmaInit	initial sigma value (step size), default is 1.0
nSigma	number of different sigmas, default is 1
tau0	number, default is 0.0. tau0 is the general multiplier.
tau	number, learning parameter for self adaption, default is 1.0. tau is the local multiplier for step sizes (for each dimension).
rho	number of parents involved in the procreation of an offspring (mixing number), default is "bi"
sel	number of selected individuals, default is 1
stratReco	Recombination operator for strategy variables. 1: none. 2: dominant/discrete (default). 3: intermediate. 4: variation of intermediate recombination.
objReco	Recombination operator for object variables. 1: none. 2: dominant/discrete (default). 3: intermediate. 4: variation of intermediate recombination.
maxGen	number of generations, stopping criterion, default is Inf
maxIter	number of iterations (function evaluations), stopping criterion, default is 100
seed	number, random seed, default is 1
noise	number, value of noise added to fitness values, default is 0.0
fName	function, fitness function, default is funSphere
lowerLimit	number, lower limit for search space, default is -1.0
upperLimit	number, upper limit for search space, default is 1.0
verbosity	defines output verbosity of the ES, default is 0
plotResult	boolean, asks if results are plotted, default is FALSE
logPlotResult	boolean, asks if plot results should be logarithmic, default is FALSE
sigmaRestart	number, value of sigma on restart, default is 0.1
preScanMult	initial population size is multiplied by this number for a pre-scan, default is 1
globalOpt	termination criterion on reaching a desired optimum value, should be a vector of length dimension (LOCATION of the optimum). Default to NULL, which means it is ignored.
...	additional parameters to be passed on to fName

---

spotLoop *Sequential Parameter Optimization Main Loop*

Description

SPOT is usually started via the function spot. However, SPOT runs can be continued (i.e., with a larger budget specified in control$funEvals) by using spotLoop. This is the main loop of SPOT iterations. It requires the user to give the same inputs as specified for spot.

Usage

```
spotLoop(x, y, fun, lower, upper, control, ...)
```

Arguments

x	are the known candidate solutions that the SPOT loop is started with, specified as a matrix. One row for each point, and one column for each optimized parameter.
y	are the corresponding observations for each solution in x, specified as a matrix. One row for each point.
fun	is the objective function. It should receive a matrix x and return a matrix y. In case the function uses external code and is noisy, an additional seed parameter may be used, see the control$seedFun argument below for details.
lower	is a vector that defines the lower boundary of search space. This determines also the dimensionality of the problem.
upper	is a vector that defines the upper boundary of search space.
control	is a list with control settings for spot. See spotControl.
...	additional parameters passed to fun.

Value

This function returns a list with:

xbest Parameters of the best found solution (matrix).

ybest Objective function value of the best found solution (matrix).

x Archive of all evaluation parameters (matrix).

y Archive of the respective objective function values (matrix).

count Number of performed objective function evaluations.

msg Message specifying the reason of termination.

modelFit The fit of the last build model, i.e., an object returned by the last call to the function specified by control$model.

Examples

```
## Most simple example: Kriging + LHS + predicted
## mean optimization (not expected improvement)
control <- list(funEvals=20)
res <- spot(,funSphere,c(-2,-3),c(1,2),control)
## now continue with larger budget
control$funEvals <- 25
res2 <- spotLoop(res$x,res$y,funSphere,c(-2,-3),c(1,2),control)
res2$xbest
res2$ybest
```

`wrapFunction` *Function Evaluation Wrapper*

Description

This is a simple wrapper that turns a function of type `y=f(x)`, where x is a vector and y is a scalar, into a function that accepts and returns matrices, as required by `spot`. Note that the wrapper essentially makes use of the apply function. This is effective, but not necessarily efficient. The wrapper is intended to make the use of spot easier, but it could be faster if the user spends some time on a more efficient vectorization of the target function.

Usage

```
wrapFunction(fun)
```

Arguments

`fun`	the function `y=f(x)` to be wrapped, with x a vector and y a numeric

Value

a function in the style of `y=f(x)`, accepting and returning a matrix

Examples

```
## example function
branin <- function (x) {
  y <- (x[2]  - 5.1/(4 * pi^2) * (x[1] ^2) + 5/pi * x[1]  - 6)^2 +
    10 * (1 - 1/(8 * pi)) * cos(x[1] ) + 10
  y
}
## vectorize / wrap
braninWrapped <-wrapFunction(branin)
## test original
branin(c(1,2))
branin(c(2,2))
branin(c(2,1))
## test wrapped
braninWrapped(matrix(c(1,2,2,2,2,1),3,2,byrow=TRUE))
```

Index