

Package ‘SPOT’

June 25, 2022

License GPL (>= 2)

Title Sequential Parameter Optimization Toolbox

Type Package

LazyLoad yes

LazyData true

LazyDataCompression gzip

Encoding UTF-8

Description A set of tools for model-based optimization and tuning of algorithms (hyperparameter tuning respectively hyperparameter optimization). 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 2.11.14

Date 2022-06-11

Depends R (>= 3.5.0)

Imports DEoptim, ggplot2, glmnet, graphics, grDevices, laGP, MASS, nloptr, plgp, plotly, rpart, randomForest, ranger, rgenoud, rsm, stats, utils

RoxygenNote 7.2.0

Suggests batchtools, car, farff, knitr, microbenchmark, rmarkdown, OpenML, party, RColorBrewer, readr, testthat

VignetteBuilder knitr

URL <https://www.spotseven.de>

NeedsCompilation no

Author Thomas Bartz-Beielstein [aut, cre]

(<https://orcid.org/0000-0002-5938-5158>),

Martin Zaeferrer [aut] (<https://orcid.org/0000-0003-2372-2092>),

Frederik Rehbach [aut] (<https://orcid.org/0000-0003-0922-8629>),

Margarita Rebolledo [ctb],
 Joerg Stork [ctb] (0000-0002-7471-3498),
 Christian Lasarczyk [ctb]

Maintainer Thomas Bartz-Beielstein <tbb@bartzundbartz.de>

Repository CRAN

Date/Publication 2022-06-25 20:00:02 UTC

R topics documented:

SPOT-package	5
buildBO	5
buildCVModel	7
buildEnsembleStack	8
buildGaussianProcess	9
buildKriging	10
buildKrigingDACE	13
buildLasso	14
buildLM	15
buildLOESS	16
buildPCA	17
buildRandomForest	18
buildRanger	19
buildrsdummy	21
buildRSM	21
buildTreeModel	22
checkArrival	23
checkFeasibilityNlopGnIngres	24
code2nat	24
dataGasSensor	25
descentSpotRSM	26
designLHD	27
designUniformRandom	28
diff0	29
doParallel	30
expectedImprovement	30
funBard	31
funBeale	32
funBox3d	32
funBranin	33
funBrownBs	34
funCosts	35
funCyclone	35
funError	37
funFreudRoth	38
funGauss	39
funGoldsteinPrice	39
funGulf	40

funHelical	41
funIshigami	42
funJennSamp	43
funMeyer	44
funMoo	45
funNoise	45
funOptimLecture	46
funPowellBs	47
funPowellS	47
funRosen	48
funRosen2	49
funShiftedSphere	50
funSoblev99	50
funSphere	51
funSring	52
getCosts	53
getMultiStartPoints	54
getNatDesignFromCoded	54
getPerformanceStats	55
getPositions	55
getPower	56
getReplicates	57
getSampleSize	57
handleNAsKrigingWorst	58
handleNAsMax	59
handleNAsMean	60
imputeY	61
infillEI	62
infillExpectedImprovement	62
init_ring	63
makeMoreFunList	64
makeSpotFunList	65
normalizeMatrix	66
normalizeMatrix2	67
obj.plgpEI	67
objectiveFunctionEvaluation	68
ocbaRanking	70
optimDE	71
optimES	72
optimGenoud	74
optimLagp	75
optimLBFGSB	76
optimLHD	77
optimNLOPTR	78
optimRSfun	79
perceptron	80
plgpEI	80
plot.spotSeverity	81

plotBestObj	83
plotData	83
plotFunction	85
plotModel	87
plotPCA	88
plotPCAVariance	89
predict.cvModel	91
predict.spotBOModel	91
prepareBestObjectiveVal	92
repeatsOCBA	92
resBench01	93
resSpot	94
resSpot2	94
ring	95
runOptim	96
runSpotBench	97
sann2spot	98
satter	98
simulate.kriging	99
simulateFunction	100
spot	102
spotAlgEs	103
spotCleanup	105
spotControl	106
spotLoop	108
spotPlotErrors	109
spotPlotPower	111
spotPlotSeverityBasic	111
spotPlotTest	112
spotPower	113
spotSeverity	114
spotSeverityBasic	115
sring	116
sringRes1	116
sringRes2	117
sringRes3	117
thetaNugget	118
thetaNuggetGradient	118
transformX	119
vmmessage	119
wrapBatchTools	120
wrapFunction	121
wrapFunctionParallel	122
wrapSystemCommand	122

Description

Sequential Parameter Optimization Toolbox

Details

SPOT uses a combination of statistical 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.

Maintainer

Thomas Bartz-Beielstein <tbb@bartzundbartz.de>

Author(s)

Thomas Bartz-Beielstein <tbb@bartzundbartz.de>, Martin Zaefferer, and F. Rehbach with contributions from: C. Lasarczyk, M. Rebolledo, Joerg Stork.

See Also

Main interface function is [spot](#).

Description

Bayesian Optimization Model Interface

Usage

```
buildBO(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: 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 L-BFGS-B 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

Value

an object of class "spotBOModel", with a predict method and a print method. Basically a list, with the options and found parameters for the model which has to be passed to the predictor function:

x	sample locations
y	observations at sample locations (see parameters)
min	min y val
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)
lambdaLower	lower boundary for log10lambda, default is -6
lambdaUpper	upper boundary for log10lambda, default is 0
dmodeltheta	vector of activity parameters
dmodellambda	regularization constant (nugget)
mu	mean mu
ssq	sigma square
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.

Gramacy, R. B. Surrogates. CRC press, 2020.

Jones, D. R., Schonlau, M., and Welch, W. J. Efficient global optimization of expensive black-box functions. Journal of Global Optimization 13, 4 (1998), 455–492.

See Also

[predict.spotBOModel](#)

Examples

```
## Reproduction of Gramacy's classic EI illustration with data from Jones et al.
## Generates Fig. 7.6 from the Gramacy book "Surrogates".
x <- c(1, 2, 3, 4, 12)
y <- c(0, -1.75, -2, -0.5, 5)
## Build B0 Model
m1 <- buildB0(x = matrix(x, ncol = 1),
  y = matrix(y, ncol=1),
  control = list(target="ei"))
xx <- seq(0, 13, length=1000)
yy <- predict(object = m1, newdata = xx)
m <- which.min(y)
fmin <- y[m]
mue <- matrix(yy$y, ncol = 1)
s2 <- matrix(yy$s, ncol = 1)
ei <- matrix(yy$ei, ncol = 1)
## Plotting the Results (similar to Fig. 7.6 in Gramacy's Surrogate book)
par(mfrow=c(1,2))
plot(x, y, pch=19, xlim=c(0,13), ylim=c(-4,9), main="predictive surface")
lines(xx, mue)
lines(xx, mue + 2*sqrt(s2), col=2, lty=2)
lines(xx, mue - 2*sqrt(s2), col=2, lty=2)
abline(h=fmin, col=3, lty=3)
legend("topleft", c("mean", "95% PI", "fmin"), lty=1:3, col=1:3, bty="n")
plot(xx, ei, type="l", col="blue", main="EI", xlab="x", ylim=c(0,max(ei)))
```

buildCVModel

buildCVModel

Description

Build a set of models trained on different folds of cross-validated data. Can be used to estimate the uncertainty of a given model type at any point.

Usage

```
buildCVModel(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.
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. This can also be changed after the model has been built, by manipulating the respective `object$target` value.
uncertaintyEstimator a character vector specifying which uncertaintyEstimator should be used. "s" or the linearlyAdapted uncertainty "sLinear". Default is "sLinear".
modellingFunction the model that shall be fitted to each data fold

Value

set of models (class `cvModel`)

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, build...`).
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 design points
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points
y <- funBranin(x)
## Create model with default settings
fit <- buildEnsembleStack(x,y)
## Predict new point
predict(fit,cbind(1,2))
## True value at location
funBranin(matrix( c(1,2), 1))
```

buildGaussianProcess *Gaussian Process Model Interface*

Description

Gaussian Process Model Interface

Usage

```
buildGaussianProcess(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. n subset size.

Value

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

Examples

```
N <- 200
x <- matrix( seq(from=-1, to = 1, length.out = N), ncol = 1)
y <- funSphere(x) + rnorm(N, 0, 0.1)
fit <- buildGaussianProcess(x,y)
## Print model parameters
print(fit)
## Predict at new location
xNew <- matrix( c(-0.1, 0.1), ncol = 1)
predict(fit, xNew)
## True value at location
t(funSphere(xNew))
```

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. Note: This can also be changed after the model has been built, by manipulating the respective object\$target value.
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 optimDE.
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 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](#).

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. $\log_{10}(dmodeltheta)$)
- dmodellambda regularization constant (nugget)
- Lambda log_10 of regularization constant (nugget) (i.e. $\log_{10}(dmodellambda)$)
- 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

```
## Create design points
set.seed(1)
x <- cbind(runif(20)*15-5,runif(20)*15)
y <- funBranin(x)
## 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
funBranin(matrix(c(1,2), 1))
##
## 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=optimDE))
## fit the model (give information about the factor variable)
fitFactor <- buildKriging(x,y,control =
list(algTheta=optimDE,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	<i>Build DACE model</i>
------------------	-------------------------

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

<code>x</code>	design matrix (sample locations), rows for each sample, columns for each variable.
<code>y</code>	vector of observations at <code>x</code>
<code>control</code>	(list), with the options for the model building procedure: <code>startTheta</code> optional start value for theta optimization, default is NULL <code>algTheta</code> algorithm used to find theta, default is <code>optimDE</code> . <code>budgetAlgTheta</code> 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. <code>nugget</code> 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. <code>regr</code> Regression function to be used: <code>regpoly0</code> (default), <code>regpoly1</code> , <code>regpoly2</code> . Can be a custom user function. <code>corr</code> Correlation function to be used: <code>corrnoisykriging</code> (default), <code>corrkriging</code> , <code>corrnoisygauss</code> , <code>corrgauss</code> , <code>correxpg</code> , <code>corrln</code> , <code>corrncubic</code> , <code>corrspherical</code> , <code>corrspline</code> . Can also be user supplied (if in the right form). <code>target</code> 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 <code>predict.kriging</code> . This can also be changed after the model has been build, by manipulating the respective <code>object\$target</code> value.

Value

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

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

Author(s)

The authors of the original DACE Matlab toolbox are Hans Bruun Nielsen, 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=corr spline))
## Print model parameters
print(fit)
```

buildLasso

Lasso Model Interface

Description

The purpose of this function is to provide an interface as required by [spot](#), to enable modeling and model-based optimization with Lasso models.

Usage

```
buildLasso(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 parameter 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 "spotLassoModel", 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 <- buildLasso(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))
```

 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

```
## Create design points
set.seed(1)
x <- cbind(runif(20)*15-5,runif(20)*15)
## Compute observations at design points (for Branin function)
y <- funBranin(x)
## Create model
fit <- buildLM(x,y)
## Print model parameters
print(fit)
## Predict at new location
predict(fit,cbind(1,2))
## True value at location
funBranin(cbind(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
```

buildPCA

buildPCA

Description

buildPCA builds principal components of given dataset. It is used inside [plotPCA](#) function to build necessary object to perform principal components analysis.

Usage

```
buildPCA(x, control = list())
```

Arguments

x	dataset of parameters to be transformed
control	control list

Value

returns a list with the following elements:

sdev the standard deviations of the principal components (i.e., the square roots of the eigenvalues of the covariance/correlation matrix, though the calculation is actually done with the singular values of the data matrix).

rotation the matrix of variable loadings (i.e., a matrix whose columns contain the eigenvectors).

x transformed matrix.

center, scale the centering and scaling used, or FALSE.

Author(s)

Alpar Gür <alpar.guer@smail.th-koeln.de>

Examples

```
#define objective function

objFun <- function(x) 2*(x[1] - 1)^2 + 5*(x[2] - 3)^2 + (10*x[3] - x[4])/3

spotConfig <-
list(types = c('numeric', 'numeric', 'numeric', 'numeric'),
funEvals = 15, #budget
noise = TRUE,
seedFun = 1,
replicated = 2,
seedSPOT = 1,
design = designLHD,
model = buildRandomForest, #surrogate model
optimizer = optimLHD, #LHD to optimize model
optimizerControl = list(funEvals=100)) #100 model evals in each step

lower <- c(-20, -20, -20, -20)
upper <- c(20, 20, 20, 20)

res <- spot(x=NULL,
fun=objFun,
lower=lower,
upper=upper,
control=spotConfig)

resPCA <- buildPCA(res$x)
```

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

```
## 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)
## Print model parameters
print(fit)
## Predict at new location
predict(fit,cbind(1,2))
## True value at location
braninFunction(c(1,2))
```

Description

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

Usage

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

Arguments

<code>x</code>	matrix of input parameters. Rows for each point, columns for each parameter.
<code>y</code>	one column matrix of observations to be modeled.
<code>control</code>	list of control parameters. These are all configuration parameters of the <code>ranger</code> function, and will be passed on to it.

Value

an object of class `spotRanger`, with a `predict` method and a `print` method. #'

Examples

```
## Create a simple training data set
testfun <- function (x) x[1]^2
x <- cbind(sort(runif(30)*2-1))
y <- as.matrix(apply(x,1,testfun))
## test data:
xt <- cbind(sort(runif(3000)*2-1))
## Example with default model (standard randomforest)
fit <- buildRanger(x,y)
yt <- predict(fit,data.frame(x=xt))
plot(xt,yt$y,type="l")
points(x,y,col="red",pch=20)
## Example with extra trees, an interpolating model
fit <- buildRanger(x,y,
  control=list(rangerArguments =
    list(replace = FALSE,
         sample.fraction=1,
         min.node.size = 1,
         splitrule = "extratrees")))
yt <- predict(fit,data.frame(x=xt))
plot(xt,yt$y,type="l")
points(x,y,col="red",pch=20)
```

buildrsdummy	<i>Build random search dummy model</i>
--------------	--

Description

This function is used to emulate uniform random search with SPOT. It is a placeholder for the surrogate model and simply returns an empty list, with class "rsdummy".

Usage

```
buildrsdummy(x, y, control)
```

Arguments

x	x (independent variables), not used.
y	y (dependent variable), not used.
control	control, not used.

buildRSM	<i>Build Response Surface Model</i>
----------	-------------------------------------

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: <i>mainEffectsOnly</i> Logical, defaults to FALSE. Set to TRUE if a model with main effects only is desired (no interactions, second order effects). <i>canonical</i> 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)
```

buildTreeModel

buildTreeModel

Description

Regression Interface This is a simple wrapper for the `rpart` function from the `rpart` package. The purpose of this function is to provide an interface as required by SPOT, to enable modeling and model-based optimization with regression trees.

Usage

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

Arguments

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

Value

an object of class `spotTreeModel`, with a `predict` method and a `print` method.

Examples

```
## Create design points
set.seed(1)
x <- cbind(runif(20)*15-5, runif(20)*15)
## Compute observations at design points (for Branin function)
y <- funBranin(x)
## Create model
fit <- buildTreeModel(x,y)
## Print model parameters
print(fit)
## Predict at new location
predict(fit,cbind(1,2))
## True value at location
funBranin(matrix( c(1,2), 1, ))
##
set.seed(123)
x <- seq(-1,1,1e-2)
y0 <- c(-10,10)
sfun0 <- stepfun(0, y0, f = 0)
y <- sfun0(x)
fit <- buildTreeModel(x,y)
# plot(fit)
# plot(x,y, type = "l")
yhat <- predict(fit, newdata = 1)
yhat$y == 10
```

checkArrival

checkArrival

Description

Calculate arrival events for S-Ring.

Usage

```
checkArrival(probNewCustomer)
```

Arguments

probNewCustomer

probability of an arrival of a new customer

Value

logical

Examples

```
checkArrival(0.5)
```

checkFeasibilityNlopGnIngres
Check feasibility for NLOPT_GN_ISRES

Description

Returns TRUE if x does satisfy ineq constraint OR no constraint function is used

Usage

```
checkFeasibilityNlopGnIngres(x, control)
```

Arguments

x	(1 x n)-matrix to be tested
control	Control list for spot and spotLoop . Generated with spotControl .

Value

logical (TRUE if feasible)

code2nat *Transform coded values to natural values*

Description

Input values from the interval from zero to one, i.e., normalized values, are mapped to the interval from a to b.

Usage

```
code2nat(x, a, b)
```

Arguments

x	matrix of m n-dimensional input values from the interval [0;1], i.e, dim(x) = m x n
a	vector of n-dimensional lower bound, i.e., length(a) = n
b	vector of n-dimensional upper bound, i.e., length(b) = n

Examples

```
x <- matrix(runif(10),2)
a <- c(-1,1,2,3,4)
b <- c(1,2,3,4,5)
R <- code2nat(x,a,b)
```

dataGasSensor	<i>Gas Sensor Data</i>
---------------	------------------------

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

Details

Two different modeling tasks are of interest for this data set: $Y \sim X_1 + X_2 + X_3 + X_4 + X_5 + X_6 + X_7 + \text{Batch} + \text{Sensor}$ and $X_1 \sim Y + X_7 + \text{Batch} + \text{Sensor}$.

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 pair-wise 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 Zaeggerer

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 replicates 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)
```

diff0

*diff0***Description**

Calculate differences

Usage

```
diff0(x)
```

Arguments

x input vector

Details

Input vector length = output vector length

Value

vector of differences

Examples

```
x <- 1:10
diff0(x)
```

doParallel	<i>Parallel execution of code, dependent on the operating system</i>
------------	--

Description

mclapply is only supported on linux and macOS. On Windows parLapply should be used. This function switches between both dependent on the operating system of the user.

Usage

```
doParallel(X, FUN, nCores = 2, ...)
```

Arguments

X	vector with arguments to parallelize over
FUN	function that shall be applied to each element of X
nCores	integer. Defines the number of cores.
...	optional arguments to FUN

expectedImprovement	<i>Expected Improvement</i>
---------------------	-----------------------------

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, $-\log_{10}(EI)$.

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
```

funBard

funBard (No. 14, More No. 8)

Description

3-dim Bard Test Function

$x_0 = (1,1,1)$ $f = 8.21487...1e-3$ $f = 17.4286...$ at $(0.8406..., -infty, -infty)$

Usage

```
funBard(x)
```

Arguments

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

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

BARD, Y. Comparison of gradient methods for the solution of nonlinear parameter estimation problems *SIAM J. Numer. Anal.* 7 (1970), 157-186.

Examples

```
x1 <- matrix(c(1,1),1,2)
funBard(x1)
```

funBeale *funBeale (No.11, More No. 5)*

Description

2-dim Beale Test Function

Usage

```
funBeale(x)
```

Arguments

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

Value

1-column matrix with resulting function values

References

Beale, E.M.L. On an interactive method of finding a local minimum of a function of more than one variable. Tech. Rep. No. 25, Statistical Techniques Research Group, Princeton Univ., Princeton, N.J., 1958.

Rosenbrock, H. (1960). An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3), 175-184. [doi:10.1093/comjnl/3.3.175](https://doi.org/10.1093/comjnl/3.3.175)

Examples

```
x1 <- matrix(c(1,1),1,2)
funBeale(x1)

res <- spot(funBeale,c(1,-1),c(5,2),control=list(funEvals=15))
plotModel(res$model)
```

funBox3d *funbox3D (No. 18, More No. 12)*

Description

Box three-dimensional Test Function

Usage

```
funBox3d(x)
```


Arguments

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

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

Box three - dimensional, (1966). A comparison of several current optimization methods, and the use of transformations in constrained problems. *The Computer Journal*, 3(3), 66-77. <https://academic.oup.com/comjnl/article/9/1/67/348150>

```
@examples x <- matrix(c(1,10,1),1) funBox3d(x)
```

```
res <- spot(,funBox3d,c(5,15,-5),c(15,5,5),control=list(funEvals=20)) # plotting the graphs plot-
Model(res$model,which=1:2) plotModel(res$model,which=2:3) plotModel(res$model,which=c(1,3))
```

funBranin

funBranin (No. 1)

Description

Branin Test Function

Usage

```
funBranin(x)
```

Arguments

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

Value

1-column matrix with resulting function values

Examples

```
x1 <- matrix(c(-pi, 12.275),1,)
funBranin(x1)
```

funBrownBs *funbrownBs* (No. 10, More No. 4)

Description

2-dim Brown badly scaled Test Function

Usage

```
funBrownBs(x)
```

Arguments

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

Details

$n=2$, $m=3$ $x_0 = (1,1)$ $f=0$ at $(1e6, 2e-6)$

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. <https://www.osti.gov/servlets/purl/6650344>

Examples

```
x1 <- matrix(c(1,1),1,2)
funBrownBs(x1)

res <- spot(, fun=funBrownBs, c(-10, -10), c(10, 10), control=list(funEvals=20))
plotModel(res$model, points = rbind(c(res$xbest[1], res$xbest[2]), c(1.098e-5, 9.106)))
```

`funCosts`*funCosts*

Description

optimWrapper for getCosts

Usage

```
funCosts(x)
```

Arguments

`x` vector: weight multiplier `sigma` and number of elevators `ne`

Details

Evaluate synthetic cost function that is based on the number of waiting customers and the number elevators

Value

fitness (costs) as matrix

Examples

```
sigma = 1
ne = 10
x <- matrix(c(sigma, ne), 1,)
funCosts(x)
```

`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(TRUE, TRUE, TRUE),
  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

<code>x</code>	vector of length at least one and up to six, specifying non-default geometrical parameters in [m]: Da, H, Dt, Ht, He, Be
<code>deterministic</code>	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).
<code>cyclone</code>	list of a default cyclone's geometrical parameters: <code>fluid\$Da</code> , <code>fluid\$H</code> , <code>fluid\$Dt</code> , <code>fluid\$Ht</code> , <code>fluid\$He</code> and <code>fluid\$Be</code>
<code>fluid</code>	list of default fluid parameters: <code>fluid\$Mu</code> , <code>fluid\$Vp</code> , <code>fluid\$Rhop</code> , <code>fluid\$Rhof</code> and <code>fluid\$Croh</code>
<code>noiseLevel</code>	list of noise levels for volume flow (<code>noiseLevel\$Vp</code>) and particle density (<code>noiseLevel\$Rhop</code>), only used if non-deterministic.
<code>model</code>	type of the model (collection efficiency only): either "Barth-Muschelknautz" or "Mothes"
<code>intervals</code>	vector specifying the particle size interval bounds.
<code>delta</code>	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 function, 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
```

funError

funError

Description

Simulate NAs, Infs, NaNs in results from objective function evaluations

Usage

```
funError(x, prob = 0.1, errorList = list(NA, Inf, NaN), outDim = 1)
```

Arguments

x	input vector or matrix of candidate solution
prob	error probability (0<prob<1). Default: 0.1
errorList	list with error types. Default: list(NA, Inf, NaN)
outDim	dimension of the output matrix (number of columns)

Details

Results from [funSphere](#) are replaced with NA, NaN, and Inf values.

Value

vector of objective function values

See Also

[is.finite](#)

Examples

```
set.seed(123)
require(SPOT)
x <- matrix(1:10, 5, 2)
y <- funError(x)
any(is.na(y))
## two-dim output
funError(x, outDim=2)
funError(x, outDim=2, prob=0.1)
```

funFreudRoth

funFreudRoth (No. 8, More No. 2)

Description

2-dim Freudstein and Roth Test Function

Usage

```
funFreudRoth(x)
```

Arguments

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

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

B. Freudstein, F., and Roth, B. (Oct. 1963). Numerical solutions of systems of nonlinear equations. *The ACM Journal*, 3(3), 550-556. <https://dl.acm.org/doi/10.1145/321186.321200>

Examples

```
x1 <- matrix(c(1,1),1,2)
funFreudRoth(x1)

# Running SPOT with 20 function evaluations with default configurations
res <- spot(funFreudRoth, c(0,0), c(10,10), control=list(funEvals=20))
plotModel(res$model)
```

funGauss	<i>funGauss (No. 15, More No. 9)</i>
----------	--------------------------------------

Description

3-dim Gaussian Test Function

Usage

```
funGauss(x)
```

Arguments

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

Value

1-column matrix with resulting function values

References

Unpublished

Examples

```
x1 <- matrix(c(1,1,1),1,3)
funGauss(x1)

res1 <- spot(,funGauss,
  c(-0.001,-0.007,-0.003),
  c(0.5,1.0,1.1),
  control=list(funEvals=15))
plotModel(res1$model, which = 1:2)
```

funGoldsteinPrice	<i>Goldstein-Price Test Function (No. 5)</i>
-------------------	--

Description

An implementation of Booker et al.'s method on a re-scaled/coded version of the 2-dim Goldstein-Price function

Usage

```
funGoldsteinPrice(x)
```

Arguments

x (m, 2)-matrix of points to evaluate with the function. Rows for points and columns for dimension.

Value

1-column matrix with resulting function values

Examples

```
x1 <- matrix(c(-pi, 12.275),1,2)
funGoldsteinPrice(x1)
```

funGulf *funGulf (No.17, More No. 11)*

Description

3-dim Gulf research and development Test Function

Usage

```
funGulf(x, m = 99)
```

Arguments

x matrix (n x 3) of points to evaluate with the function. Rows for points and columns for dimension. Values should be larger than 0.

m additional parameter: . The Gulf function supports an additional parameter m in the range from 3 to 100

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

Examples

```
x1 <- matrix(c(50,25,1.5),1,3)
funGulf(x1)

funGulf(x1,m=50)

resGulf <- spot(,funGulf,c(0,0,0),c(100,50,5))
resGulf$xbest
resGulf$ybest
plotModel(resGulf$model, which=1:2)
plotModel(resGulf$model, which=2:3)

# x0 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.
x0 = matrix(c(5,2.5,0.15),1,3)
resGulf <- spot(x0,funGulf,c(0,0,0),c(100,50,5))
resGulf$xbest
resGulf$ybest
```

funHelical

funHelical (No. 13, More No. 7)

Description

3-dim Helical Test Function

Usage

```
funHelical(x)
```

Arguments

x matrix (n x 3)-dim of points to evaluate with the function. Rows for points and columns for dimension.

Value

1-column matrix with resulting function values

References

- More', J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. [doi:10.1145/355934.355936](https://doi.org/10.1145/355934.355936)
- Fletcher, R., and Powell, M. J. (1963). A rapidly convergent descent method for minimization. *The Computer Journal*, 6(2), 163-168. [doi:10.1093/comjnl/6.2.163](https://doi.org/10.1093/comjnl/6.2.163)

Examples

```
x1 <- matrix(c(1,1,1),1,)
funHelical(x1)
res <- spot(funHelical,c(-40,-40,-40),c(40,40,40),control=list(funEvals=20))
plotModel(res$model,which=c(1,2),type="persp",border="NA")
plotModel(res$model,which=c(2,3),type="persp",border="NA")
plotModel(res$model,which=c(1,3),type="persp",border="NA")
plotModel(res$model, which=c(1,2))
plotModel(res$model, which=c(1,3))
plotModel(res$model, which=c(2,3))
```

funIshigami

Ishigami Test Function (No. 7)

Description

An implementation of the 3-dim Ishigami function.

$$f(x) = \sin(x_1) + a \sin^2(x_2) + b x_3^4 \sin(x_1)$$

The Ishigami function of Ishigami & Homma (1990) is used as an example for uncertainty and sensitivity analysis methods, because it exhibits strong nonlinearity and nonmonotonicity. It also has a peculiar dependence on x_3 , as described by Sobol' & Levitan (1999). The independent distributions of the input random variables are usually: $x_i \sim \text{Uniform}[-\pi, \pi]$, for all $i = 1, 2, 3$.

Usage

```
funIshigami(x, a = 7, b = 0.1)
```

Arguments

x	(m,3)-matrix of points to evaluate with the function. Values should be $\geq -\pi$ and $\leq \pi$, i.e., x_i in $[-\pi, \pi]$.
a	coefficient (optional), with default value 7
b	coefficient (optional), with default value 0.1

Value

1-column matrix with resulting function values

References

Ishigami, T., & Homma, T. (1990, December). An importance quantification technique in uncertainty analysis for computer models. In *Uncertainty Modeling and Analysis, 1990. Proceedings., First International Symposium on* (pp. 398-403). IEEE.

Sobol', I. M., & Levitan, Y. L. (1999). On the use of variance reducing multipliers in Monte Carlo computations of a global sensitivity index. *Computer Physics Communications*, 117(1), 52-61.

Examples

```
x1 <- matrix(c(-pi, 0, pi),1,)  
funIshigami(x1)
```

funJennSamp

funJennSamp (No. 12, More No 6)

Description

2-dim Jennrich and Sampson Function Test Function

Usage

```
funJennSamp(x)
```

Arguments

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

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., & Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

Jennrich, R.I., and Sampson (1968). Application of stepwise regression to nonlinear estimation. *Technometrics*, 3(3), 63-72. <https://www.tandfonline.com/doi/abs/10.1080/00401706.1968.10490535>

Examples

```
x1 <- matrix(c(1,1),1,)  
funJennSamp(x1)  
  
res <- spot(,funJennSamp,c(0,0),c(0.3,0.3))  
plotModel(res$model)
```

funMeyer

funMeyer (No. 16, More No. 10)

Description

Meyer 3-dim Test Function

Usage

```
funMeyer(x)
```

Arguments

`x` matrix (dim 1x3) of points to evaluate with the function. Rows for points and columns for dimension.

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

Examples

```
x1 <- matrix(c(1,1,1),1,3)
funMeyer(x1)

set.seed(13)
resMeyer <- spot(matrix(c(0.02,4000,250),1,3),
  funMeyer,c(0,1000,200),c(3,8000,500),
  control= list(funEvals=15))
resMeyer$xbest
resMeyer$ybest
print("Model with parameters")
plotModel(resMeyer$model)
plotModel(resMeyer$model,which=2:3)
```

`funMoo`*funMoo*

Description

Multi-objective Test Function

Usage

```
funMoo(x)
```

Arguments

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

Details

`funMultiObjectiveOptimization`

A multiobjective dummy testfunction

Value

n-column matrix with resulting function values

Examples

```
x1 <- matrix(c(-pi, 12.275),1,)  
funMoo(x1)  
x <- matrix(c(1,1,2), ncol=3 )
```

`funNoise`*funNoise*

Description

noise function

Usage

```
funNoise(x, fun = funSphere, mean = 0, sigma = 1)
```

Arguments

x	input matrix of candidate solution
fun	objective function. Default: funSphere
mean	error mean. Default: 1
sigma	error sd. Default: 1

Value

vector of noisy objective function values

Examples

```
set.seed(123)
require(SPOT)
x <- matrix(1:10, 5, 2)
funNoise(x)
```

funOptimLecture *funOptimLecture*

Description

A testfunction used in the optimizaton lecture of the AIT Masters course at TH Koeln

Usage

```
funOptimLecture(vec)
```

Arguments

vec	input vector or matrix of candidate solution
-----	--

Value

vector of objective function values

funPowellBs *funPowellBs (No. 9, More No. 3)*

Description

2-dim Powell Badly Scaled Test Function

Usage

```
funPowellBs(x)
```

Arguments

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

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. [doi:10.1145/355934.355936](https://doi.org/10.1145/355934.355936)

Powell, M.J.D. (1970). A hybrid method for nonlinear equations. In Numerical methods for Non-linear Algebraic Equations, P. Rabinowitz (Ed), *Gordon & Breach, New York.*, 3(3), 87-114.

Examples

```
x1 <- matrix(c(-1,1),1,1)
funPowellBs(x1)

# Running SPOT with 20 function evaluations with default configurations
res <- spot(fun=funPowellBs,c(-10,-10),c(10,10),control=list(funEvals=20))
plotModel(res$model, points = rbind(c(res$xbest[1], res$xbest[2]),c(1.098e-5,9.106)))
```

funPowellS *funPowellS (No. 19, More No. 13)*

Description

Powell's 4-dim Test Function

Usage

```
funPowellS(x)
```

Arguments

x matrix (dim 1x4) of points to evaluate with the function. Rows for points and columns for dimension.

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. Trond Steihaug and Sara Suleiman Global convergence and the Powell singular function *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936 <http://owos.gm.fh-koeln.de:8055/bartz/optimization-ait-master-2020/blob/master/Jupyter.d/Exercise-VIIa.ipynb> <http://bab10.bartzandbartz.de:8033/bartzbeielstein/bab-optimization-ait-master-/blob/master/Jupyter.d/01spotNutshell.ipynb> <https://www.mat.univie.ac.at/~neum/glopt/bounds.html>

Powell's Test function, M. J. D. Powell, 1962 An automatic method for finding the local minimum of a function. *The Computer Journal*, 3(3), 175-184. <https://www.sfu.ca/~ssurjano/powell.html>

Examples

```
x1 <- matrix(c(0,0,0,0),1,)
funPowells(x1)
x2 <- matrix(c(3,-1,0,1),1,)
funPowells(x2)
x3 <- matrix(c(0,0,0,-2),1,)
funPowells(x3)
# optimization run with SPOT and 15 evaluations
res_fun <- spot(,funPowells,c(-4,-4,-4,-4 ),c(5,5,5,5),control=list(funEvals=15))
res_fun
```

funRosen

funRosen (No. 2, More No. 1)

Description

Rosenbrock Test Function

Usage

```
funRosen(x)
```

Arguments

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

Value

1-column matrix with resulting function values

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

Rosenbrock, H. (1960). An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3), 175-184. doi:10.1093/comjnl/3.3.175

Examples

```
x1 <- matrix(c(1,1),1,)  
funRosen(x1)
```

funRosen2

funRosen2 (No. 2a)

Description

Rosenbrock Test Function (2-dim)

Usage

```
funRosen2(x)
```

Arguments

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

Value

1-column matrix with resulting function values

Examples

```
x1 <- matrix(c(-pi, 12.275),1,)  
funRosen2(x1)
```


Moon et al. (2012) scale the output to have a variance of 100. For $d = 20$, they use three different b-vectors: (2, 1.95, 1.9, 1.85, 1.8, 1.75, 1.7, 1.65, 0.4228, 0.3077, 0.2169, 0.1471, 0.0951, 0.0577, 0.0323, 0.0161, 0.0068, 0.0021, 0.0004, 0), (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), and (2.6795, 2.2289, 1.8351, 1.4938, 1.2004, 0.9507, 0.7406, 0.5659, 0.4228, 0.3077, 0.2169, 0.1471, 0.0951, 0.0577, 0.0323, 0.0161, 0.0068, 0.0021, 0.0004, 0).

The generally used value of c_0 is $c_0 = 0$. The function is evaluated on x_i in $[0, 1]$, for all $i = 1, \dots, d$.

Usage

```
funSoblev99(x, b = c(rep(0.6, 10), rep(0.4, 10)), c0 = 0)
```

Arguments

<code>x</code>	($m, 2$)-matrix of points to evaluate with the function. Values should be ≥ 0 and ≤ 1 , i.e., x_i in $[0, 1]$.
<code>b</code>	d -dimensional vector (optional), with default value $b = c(0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4)$ (when $d \leq 20$)
<code>c0</code>	constant term (optional), with default value 0

Value

1-column matrix with resulting function values

References

Moon, H., Dean, A. M., & Santner, T. J. (2012). Two-stage sensitivity-based group screening in computer experiments. *Technometrics*, 54(4), 376-387.

Sobol', I. M., & Levitan, Y. L. (1999). On the use of variance reducing multipliers in Monte Carlo computations of a global sensitivity index. *Computer Physics Communications*, 117(1), 52-61.

Examples

```
x1 <- matrix(c(-pi, 12.275), 1, 2)
funSoblev99(x1)
```

funSphere

funSphere (No. 3)

Description

Sphere Test Function

Usage

```
funSphere(x)
```

Arguments

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

Value

1-column matrix with resulting function values

See Also

[funShiftedSphere](#)

Examples

```
x1 <- matrix(c(-pi, 12.275),1,)
funSphere(x1)
```

funSring

funSring

Description

wrapper for [sring](#)

Usage

```
funSring(x, opt = list(), ...)
```

Arguments

x perceptron weights

opt list of optional parameters, e.g.,

- nElevators number of elevators
- probNewCustomer probability pf a customer arrival
- nIterations Number of itertions
- randomSeed random seed

... additional parameters

Value

fitness (matrix with one column)

Examples

```
set.seed(123)
numberStates = 200
sigma = 1
x = matrix( rnorm(n = 2*numberStates, 1, sigma), 1, )
funSring(x)
```

getCosts

getCosts

Description

Evaluate synthetic cost function that is based on the number of waiting customers and the number elevators

Usage

```
getCosts(x, ...)
```

Arguments

`x` vector with sigma weight multiplier and ne number of elevators
`...` optional parameters passed to funSring

Details

Note: To accelerate testing, nIterations was set to 1e3 (instead of 1e6)

Value

fitness (costs)

Examples

```
set.seed(123)
sigma = 1
ne = 10
x <- c(sigma, ne)
getCosts(x)
```

getMultiStartPoints *Get Multi Start Points*

Description

Determine multi start points for optimization on the surrogate. Combines the current best with additional random starting points for optimization on the surrogate.

Usage

```
getMultiStartPoints(x, y, control)
```

Arguments

x	matrix of design points
y	matrix of function values (f(x))
control	Control list for spot and spotLoop . Generated with spotControl .

Value

x0 matrix of restart points

getNatDesignFromCoded *Get natural parameter values from coded +-1 representation*

Description

For given lower and upper bounds, a and b, respectively, coded input values are mapped to their natural values

Usage

```
getNatDesignFromCoded(x, a, b)
```

Arguments

x	(n,m)-dim matrix of coded values, i.e., lower values are coded as -1, upper values as +1.
a	m-dim vector of lower bounds (natural values)
b	m-dim vector of upper bounds (natural values)

Examples

```
x <- matrix(rep(-1,2),1,)
lower <- c(-10,-10)
upper <- c(10,10)
getNatDesignFromCoded(x, a = lower, b=upper)
```

getPerformanceStats *get performance stats*

Description

determines mean performance

Usage

```
getPerformanceStats(x, y)
```

Arguments

x matrix of n solutions (usually a (nxd)-matrix, where d is the problem dimension)
y matrix with objective values (usually a (nx1)-matrix)

Details

further stats will be added

Examples

```
x <- matrix(1:10, ncol=2, byrow=TRUE)
y1 <- funSphere(x) +1
y2 <- funSphere(x) -1
x <- rbind(x,x)
y <- rbind(y1, y2)
M <- getPerformanceStats(x,y)
```

getPosition *get impute positions*

Description

Determines positions in a vectors that fulfill criteria defined by a list of criteria, e.g., is.na.

Usage

```
getPosition(y, imputeCriteriaFuns = list(is.na, is.infinite, is.nan))
```

Arguments

y The vector of numerics from which NA/Inf values should be removed
imputeCriteriaFuns list criteria functions specified via imputeCriteriaFuns in [spotControl](#).
 Default: list(is.na, is.infinite, is.nan).

Value

p vector of positions that fulfill one of the criteria

Examples

```
imputeCriteriaFuns <- list(is.na, is.infinite, is.nan)
y <- c(1,2,Inf,4,NA,6)
p <- getPositions(y, imputeCriteriaFuns)
```

getPower

getPower

Description

Implements basic power calculations in R See also: <https://www.cyclismo.org/tutorial/R/power.html>

Usage

```
getPower(mu0, mu1, n, sigma, alpha, tdist = FALSE, alternative = "greater")
```

Arguments

mu0	mean value of the null hypothesis (usually referred to as H0)
mu1	mean value of the alternative hypothesis (usually referred to as H1)
n	sample size
sigma	sample s.d.
alpha	error
tdist	logical. Use Student t Distribution. Default: FALSE
alternative	a character string specifying the alternative hypothesis, must be one of "two.sided", "greater" (default) or "less".

Examples

```
## Power should be approx. 0.9183621:
getPower(mu0=5, mu1=6.5, n=20, sigma=2, alpha=0.05, tdist = FALSE,
alternative = "two.sided")
## Power should be approx. 0.8887417:
getPower(mu0=5, mu1=6.5, n=20, sigma=2, alpha=0.05, tdist = TRUE,
alternative = "two.sided")
## Compare with results from power.t.test
powerVal <- power.t.test(n=20, delta=1.5, sd=2, sig.level=0.05, type="one.sample",
alternative="two.sided",strict = TRUE)
powerVal$power
```

getReplicates	<i>get number of replicates</i>
---------------	---------------------------------

Description

determine how often appears x in X

Usage

```
getReplicates(x, X)
```

Arguments

x	row vector
X	matrix

Details

can be used to determine the number of replicates/repeated evaluations of a solution x

Examples

```
k <- 2
n <- 4
A <- matrix(1:(k*n),n,k, byrow = TRUE)
X <- rbind(A,A,A)
x <- A[1,]
## should be 3:
getReplicates(x,X)

## U has unique entries
U <- X[!duplicated(X), ]
## should be 1:
getReplicates(x,U)
```

getSampleSize	<i>getSampleSize</i>
---------------	----------------------

Description

Implements sample size calculations in R See also: <https://www.cyclismo.org/tutorial/R/power.html> and https://influentialpoints.com/Training/statistical_power_and_sample_size.htm

Usage

```
getSampleSize(mu0, mu1, alpha, beta, sigma, alternative = "greater")
```

Arguments

<code>mu0</code>	mean value of the null hypothesis (usually referred to as H0)
<code>mu1</code>	mean value of the alternative hypothesis (usually referred to as H1)
<code>alpha</code>	type I error
<code>beta</code>	type II error
<code>sigma</code>	sample s.d.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided", "greater" (default) or "less".

Value

`n` number of required samples in each arm of a trial. Note: total number of samples is $2*n$.

Examples

```
getSampleSize(mu0 = 0, mu1 = 200, alpha=0.05, beta=0.2, sigma=450,
alternative="two.sided")
getSampleSize(mu0 = 8.72, mu1 = 8.72*1.1, alpha=0.05, beta=0.2, sigma=1.3825,
alternative="greater")
getSampleSize(mu0 = 8.72, mu1 = 8.72*1.1, alpha=0.05, beta=0.2, sigma=1.3825,
alternative="two.sided")
```

`handleNAsKrigingWorst` *handleNAsKrigingWorst*

Description

Remove NAs from a vector by replacing them with a penalized Kriging-based expectation

Usage

```
handleNAsKrigingWorst(
  x,
  y,
  penaltyImputation = 3,
  imputeCriteriaFuns = list(is.na, is.infinite, is.nan)
)
```

Arguments

x The x values from which y was calculated
y The vector of numerics from which the NAs should be removed
penaltyImputation multiplier for sPredicted (penalty term). Default: 3.
imputeCriteriaFuns list criteria functions specified via imputeCriteriaFuns in [spotControl](#). Default: list(is.na, is.infinite, is.nan).

Value

y The imputed vector w/o NA and w/o Inf values.

Examples

```

imputeCriteriaFuns <- list(is.na, is.infinite, is.nan)
x <- matrix(runif(20), ncol = 2)
y <- funSphere(x)
y[3] <- NA
y[5] <- Inf
plot(y, type="b")
print(y)
y1 <- handleNAsKrigingWorst(x=x, y=y, imputeCriteriaFuns=imputeCriteriaFuns)
print(y1)
points(3, y1[3], type="b", col="red")
points(5, y1[5], type="b", col="red")

```

 handleNAsMax

handleNAsMax

Description

Remove NAs from a vector by replacing them by the current max + p*s.d., where p denotes a penalty term.

Usage

```

handleNAsMax(
  x,
  y = NULL,
  imputeCriteriaFuns = list(is.na, is.infinite, is.nan),
  penaltyImputation = 3
)

```

Arguments

x The x values from which y was calculated, not used here

y The vector of numerics from which the NAs should be removed

imputeCriteriaFuns list criteria functions specified via imputeCriteriaFuns in [spotControl](#).
Default: list(is.na, is.infinite, is.nan).

penaltyImputation penalty used for imputed values

Value

y The cleaned vector

Examples

```
vecWithNAs <- c(-1, 0, 1, NA, 3, Inf, 5, NA)
control <- spotControl(dim=length(vecWithNAs))
print(vecWithNAs)
print(handleNAsMax(y=vecWithNAs,
                   imputeCriteriaFuns= control$Imputation$imputeCriteriaFuns))
```

handleNAsMean

handleNAsMean

Description

Remove NAs from a vector by replacing them by the sample mean.

Usage

```
handleNAsMean(
  x,
  y = NULL,
  imputeCriteriaFuns = list(is.na, is.infinite, is.nan),
  penaltyImputation = 3
)
```

Arguments

x The x values from which y was calculated, not used here

y The vector of numerics from which the NAs should be removed

imputeCriteriaFuns list criteria functions specified via imputeCriteriaFuns in [spotControl](#).
Default: list(is.na, is.infinite, is.nan).

penaltyImputation penalty used for imputed values

Value

y The cleaned vector

Examples

```
vecWithNAs <- c(-1, 0, 1, NA, 3, Inf, 5, NA)
control <- spotControl(dim=length(vecWithNAs))
print(vecWithNAs)
print(handleNAsMean(y=vecWithNAs,
                    imputeCriteriaFuns= control$Imputation$imputeCriteriaFuns))
```

imputeY

Impute NAs and Inf in y

Description

Impute NAs and Inf in y

Usage

```
imputeY(x, y, control)
```

Arguments

x The x values from which y was calculated
y The vector of numerics from which NA/Inf values should be removed
control [spot](#) control list. See also [spotControl](#).

Value

y The imputed vector w/o NA and w/o Inf values.

Examples

```
x <- matrix(runif(10), ncol=2, nrow=5)
y <- funSphere(x)
y[1] <- NA
control <- spotControl(dimension = 2)
# no imputation function, i.e, w/o imputation
imputeY(x=x, y=y, control=control)
# with imputation
control$Imputation$handleNAsMethod <- handleNAsKrigingWorst
y <- imputeY(x=x, y=y, control=control)
# no imputation required:
imputeY(x=x, y=y, control=control)
```

`infillEI`*Expected Improvement Infill Criterion*

Description

Compute the negative 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. Expected Improvement infill criterion that can be passed to `control$modelControl$infillCriterion` in order to be used during the optimization in SPOT. Parameters dont have to be specified as this function is ment to be internally by SPOT.

Usage

```
infillEI(predictionList, model)
```

Arguments

`predictionList` The results of a `predict.model` call
`model` The surrogate model which was used for the prediction

Value

numeric vector, expected improvement results

Examples

```
spot(,funSphere,c(-2,-3),c(1,2), control =  
  list(infillCriterion = infillEI, modelControl = list(target = c("y", "s"))))
```

`infillExpectedImprovement`*infillExpectedImprovement*

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. Expected Improvement infill criterion that can be passed to `control$modelControl$infillCriterion` in order to be used during the optimization in SPOT. Parameters dont have to be specified as this function is ment to be internally by SPOT.

Usage

```
infillExpectedImprovement(predictionList, model)
```

Arguments

predictionList The results of a predict.model call
 model The surrogate model which was used for the prediction

Value

numeric vector, expected improvement results

Examples

```
spot(,funSphere,c(-2,-3),c(1,2), control =  

  list(infillCriterion = infillExpectedImprovement, modelControl = list(target = c("y","s"))))
```

init_ring	<i>init_ring</i>
-----------	------------------

Description

Initialize ring parameters: generate arrival probabilities for S-Ring. - set beginning states to 0 and initialize random customer states and nElevators - nStates = (number of floors * 2) - 2. For example for 4 floors, its 6 states because the upper and lower state have only one direction and all other have 2 (UP and DOWN)

Usage

```
init_ring(params)
```

Arguments

params list of
 randomSeed random seed
 nStates number of S-Ring states
 nElevators number of elevators
 probNewCustomer probability pf a customer arrival
 counter Counter: number of waiting customers
 sElevator Vector representing elevators (s)
 sCustomer Vector representing customers (c)
 currentState Current state that is calculated
 nextState Next state that is calculated
 nWeights Number of weights for the perceptron (= 2 * nStates)

Value

list (params) of

- randomSeed random seed
- nStates number of S-Ring states
- nElevators number of elevators
- probNewCustomer probability pf a customer arrival
- counter Counter: number of waiting customers
- sElevator Vector representing elevators (s)
- sCustomer Vector representing customers (c)
- currentState Current state that is calculated
- nextState Next state that is calculated
- nWeights Number of weights for the perceptron ($= 2 * nStates$)

Examples

```

params <-list(sElevator=NULL,
  sCustomer=NULL,
  currentState=NULL,
  nextState=NULL,
  counter=NULL,
  nStates=12,
  nElevators=2,
  probNewCustomer=0.1,
  weightsPerceptron=rep(0.1, 24),
  nWeights=NULL,
  nIterations=100,
  randomSeed=1234)

init_ring(params)

```

makeMoreFunList

makeMoreFunList

Description

Generate a list of benchmark functions. Based on the More(1981) paper. Contains the first 13 function from the paper. Function numbers are the same as in the paper.

Usage

```
makeMoreFunList(vector2Matrix = TRUE)
```


Arguments

vector2Matrix logical. Convert vector input to matrix. Default: TRUE, so it can be used with [optim](#).

Value

list of functions with starting points and optimum points.

References

More, J. J., Garbow, B. S., and Hillstom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41. doi:10.1145/355934.355936

Examples

```
# Generate function list.
# Here we use the default setting \code{vector2Matrix = TRUE},
# so the function list can be passed to \code{\link[stats]{optim}}.

f1 <- makeMoreFunList()
optim(par=c(-1.2,1), fn=f1$funList[[1]])
optim(par=f1$startPointList[[1]], fn=f1$funList[[1]])$value
optim(par=f1$startPointList[[1]], fn=f1$funList[[1]],NULL, method = "CG", hessian = FALSE)$value
optim(f1$startPointList[[1]], f1$funList[[1]],NULL, method = "BFGS", hessian = FALSE)$value
optim(f1$startPointList[[1]], f1$funList[[1]],NULL, method = "L-BFGS-B", hessian = FALSE)$value
```

makeSpotFunList	<i>makeSpotFunList</i>
-----------------	------------------------

Description

Generate a list of spot functions

Usage

```
makeSpotFunList(vector2Matrix = TRUE)
```

Arguments

vector2Matrix logical. Convert vector input to matrix. Default: TRUE, so it can be used with [optim](#).

Value

list of functions

Examples

```
fr <- makeSpotFunList()
optim(c(-1.2,1), fr[[1]])
```

normalizeMatrix	<i>Normalize design matrix</i>
-----------------	--------------------------------

Description

Normalize design by using minimum and maximum of the design values for input space. Each column has entries in the range from ymin to ymax.

Usage

```
normalizeMatrix(x, ymin, ymax, MARGIN = 2)
```

Arguments

x	design matrix in input space
ymin	minimum vector of normalized space
ymax	maximum vector of normalized space
MARGIN	a vector giving the subscripts which the function will be applied over. 1 indicates rows, 2 indicates columns. Default: 2.

Value

list with the following entries:

y	normalized design matrix in the range [ymin, ymax]
xmin	min in each column
xmax	max in each column

See Also

[buildKriging](#)

Examples

```
set.seed(1)
x <- matrix(c(rep(1,3), rep(2,3),rep(3,3), rep(4,3)),3,4)
## columnwise:
normalizeMatrix(x, ymin=0, ymax=1)
## rowwise
normalizeMatrix(x, ymin=0, ymax=1, MARGIN=1)
# rows with identical values are mapped to the mean:
x <- matrix(rep(0,4),2,2)
normalizeMatrix(x, ymin=0, ymax=1)
```

normalizeMatrix2	<i>Normalize design 2</i>
------------------	---------------------------

Description

Normalize design with given maximum and minimum in input space. Supportive function for Kriging model, not to be used directly.

Usage

```
normalizeMatrix2(x, ymin, ymax, xmin, xmax)
```

Arguments

x	design matrix in input space (n rows for each point, k columns for each parameter)
ymin	minimum vector of normalized space
ymax	maximum vector of normalized space
xmin	minimum vector of input space
xmax	maximum vector of input space

Value

normalized design matrix

See Also

[buildKriging](#)

obj.plgpEI	<i>Wrapper for Expected improvement (Gramacy)</i>
------------	---

Description

Wrapper for Expected improvement (Gramacy)

Usage

```
obj.plgpEI(x, fmin, gpi, pred = predGPsep)
```

Arguments

x	matrix of points to calculate EI
fmin	best function value (y) so far
gpi	Gaussian process C-side object
pred	prediction model. Default: predGPsep

Value

negative expected improvement

See Also

[plgpEI](#).

Examples

```
library(laGP)
library(plgp)

ninit <- 12
dim <- 2
X <- designLHD(,rep(0,dim), rep(1,dim), control=list(size=ninit))
y <- funGoldsteinPrice(X)
m <- which.min(y)
ymin <- y[m]
start <- matrix(X[m,], nrow =1)

## Build laGP model
gpi <- newGPsep(X, y, d=0.1, g=1e-8, dK=TRUE)
da <- darg(list(mle=TRUE, max=0.5), designLHD(,rep(0,dim), rep(1,dim), control=list(size=1000)))
mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
res <- optim(start[1,], obj.plgpEI, method="L-BFGS-B", lower=0, upper=1,
gpi=gpi, pred=predGPsep, fmin=ymin)
xnew <- c(res$par, -res$value)
print(xnew)
deleteGPsep(gpi)
```

objectiveFunctionEvaluation

objectiveFunctionEvaluation Objective Function Evaluation

Description

This function handles the evaluation of the objective function in [spot](#). This includes handling of the random number generator stream, variable transformations ([transformX](#)) as well as the actual evaluation.

Usage

```
objectiveFunctionEvaluation(x = NULL, xnew, fun, control = list(), ...)
```

Arguments

x	matrix of already known solutions, to determine whether RNG seeds for new solutions need to be incremented.
xnew	matrix of new solutions.
fun	objective function to evaluate the solutions in xnew.
control	control list with the following entries: seedFun: initial seed to be used for the random number generator seed. Set to NA to avoid using a fixed seed. noise: logical parameter specifying whether the target function is noisy. verbosity: verbosity. Default: 0. transformFun: transformation functions applied to xnew. See transformX
...	parameters passed to fun.

Value

the matrix ynew, which are the observations for fun(xnew)

See Also

[spot](#) for more details on the parameters, e.g., fun
[transformX](#)
[spotControl](#)

Examples

```
## 1) without noise
x <- NULL
xnew <- matrix(1:10, ncol=2)
fun <- funSphere
control <- spotControl(dim(xnew)[2])
control$verbosity <- 0
objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
##
fun <- funMoo
objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
## 2) with noise
fun = function(x){funSphere(x) + rnorm(nrow(x))}
control$noise <- TRUE
objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
## 3) known solutions
x <- matrix(11:20, ncol=2)
xnew <- matrix(1:10, ncol=2)
fun <- funSphere
objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
## 4) known solutions with noise and repeats
x <- matrix(1:20, ncol=2, byrow=TRUE)
xnew <- matrix(1:10, ncol=2, byrow=TRUE)
```

```

fun = function(x){funSphere(x) + rnorm(nrow(x))}
objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
## 5) identical solutions with noise and repeats
x <- matrix(1:10, ncol=2, byrow=TRUE)
xnew <- x
fun = function(x){funSphere(x) + rnorm(nrow(x))}
y <- objectiveFunctionEvaluation(x=NULL, xnew=x, fun=fun, control=control)
y1 <- objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
y2 <- objectiveFunctionEvaluation(x=NULL, xnew=xnew, fun=fun, control=control)
print(cbind(x, y))
print(cbind(xnew, y1))
print(cbind(xnew, y2))
identical(y, y1) # FALSE
identical(y, y2) # TRUE
## 6) known solutions with noise and repeats. function sets seed
x <- matrix(1:20, ncol=2, byrow=TRUE)
xnew <- matrix(1:10, ncol=2, byrow=TRUE)
fun <- function(x,seed){
  set.seed(seed)
  funSphere(x)+rnorm(nrow(x))}
control$seedFun <- 1
y1 <- objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
y2 <- objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
identical(y1, y2) # TRUE
control$seedFun <- 2
y3 <- objectiveFunctionEvaluation(x=x, xnew=xnew, fun=fun, control=control)
identical(y1,y3) # FALSE
## 7) spot examples:
res1a <- spot(, function(x, seed){set.seed(seed);funSphere(x)+rnorm(nrow(x))},
c(-2, -3),c(1, 2),control=list(funEvals=25,noise=TRUE, seedFun=1))
res1b <- spot(, function(x, seed){set.seed(seed);funSphere(x)+rnorm(nrow(x))},
c(-2, -3),c(1, 2),control=list(funEvals=25,noise=TRUE, seedFun=1))
res2 <- spot(, function(x, seed){set.seed(seed);funSphere(x)+rnorm(nrow(x))},
c(-2, -3),c(1, 2),control=list(funEvals=25,noise=TRUE, seedFun=2))
sprintf("Should be equal: %f = %f. Should be different: %f", res1a$ybest,
res1b$ybest, res2$ybest)

```

ocbaRanking

ocbaRanking

Description

Return the ocba ranking (xbest, ybest) for noisy optimization

Usage

```
ocbaRanking(x, y, fun, control, ...)
```

Arguments

x	matrix of x values
y	matrix of y values, one dimensional!
fun	objective function
control	control list, see spotControl
...	additional arguments to fun

Details

Based on [repeatsOCBA](#)

Value

(x,y) matrix of sorted (by y) values. In case of noise are these values aggregated (y-mean) values.

 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
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$ybest
optimDE(x = matrix(rep(1,6), 3, 2),lower = c(-10,-20),upper=c(20,8),fun = funSphere,
  control = list(funEvals=100, populationSize=20))
#Compare to DEoptim:
require(DEoptim)
set.seed(1234)
DEoptim(function(x){funRosen(matrix(x,1))}, lower=c(-10,-10), upper=c(10,10),
  DEoptim.control(strategy = 2,bs = FALSE, N = 20, itermax = 28, CR = 0.7, F = 1.2,
  trace = FALSE, p = 0.2, c = 0, reltol = sqrt(.Machine$double.eps), steptol = 200 ))
set.seed(1234)
optimDE(fun=funRosen, lower=c(-10,-10), upper= c(10,10),
  control = list( populationSize = 20, funEvals = 580, F = 1.2, CR = 0.7))
```

 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
```

 optimLagp

Interface to minimization based on Gramacy's lagp package

Description

Implements Gramacy's plgp package based optimization using expected improvement. Example from chapter 7 in the surrogate book.

Usage

```
optimLagp(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
```

 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

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

Value

	list, with elements
<code>x</code>	NA, not used
<code>y</code>	NA, not used
<code>xbest</code>	best solution
<code>ybest</code>	best observation
<code>count</code>	number of evaluations of <code>fun</code> (estimated from the more complicated "counts" variable returned by <code>optim</code>)
<code>message</code>	termination message returned by <code>optim</code>

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

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

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

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

```
##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
```

optimRSfun

Random search surrogate-optimizer

Description

This function is used to emulate uniform random search with SPOT. It is used as the optimizer that searches for new candidates. It returns a single uniform random sample within the given lower and upper bounds of the search space.

Usage

```
optimRSfun(x, fun, lower, upper, control, ...)
```

Arguments

x	start guess, not used.
fun	objective function to be evaluated via random search.
lower	bound on the independent variables (search space).
upper	bound on the independent variables (search space).
control	not used.
...	additional arguments, not used.

Value

list

perceptron	<i>perceptron</i>
------------	-------------------

Description

Perceptron to calculate decisions

Usage

```
perceptron(currentState, nStates, sElevator, sCustomer, weightsPerceptron)
```

Arguments

currentState	current state for decision (num)
nStates	number of states (int)
sElevator	elevators vector (logical)
sCustomer	customer vector (logical)
weightsPerceptron	Weight vector (num)

Details

Number of weights in NN controller is $2 \times nStates$, for each state (sElevator/sCustomer) there is one input

Value

logical pass or take decision

plgpEI	<i>Expected improvement (Gramacy)</i>
--------	---------------------------------------

Description

Expected improvement (Gramacy)

Usage

```
plgpEI(gpi, x, fmin, pred = predGPsep)
```

Arguments

gpi	Gaussian process C-side object
x	matrix of points to calculate EI
fmin	best function value (y) so far
pred	prediction model. Default: predGPsep

Value

ei expected improvement

Examples

```

library(laGP)
library(plgp)

ninit <- 12
dim <- 2
X <- designLHD(,rep(0,dim), rep(1,dim), control=list(size=ninit))
y <- funGoldsteinPrice(X)
m <- which.min(y)
ymin <- y[m]
start <- matrix(X[m,], nrow =1)

## 1. Build SPOT BO Model
m1 <- buildBO(x = X, y = y, control = list(target="ei"))
yy <- predict(object = m1, newdata = start)
ei1 <- matrix(yy$ei, ncol = 1)
## Show mue and s
mue <- matrix(yy$y, ncol = 1)
s2 <- matrix(yy$s, ncol = 1)

## 2. Build laGP model
gpi <- newGPsep(X, y, d=0.1, g=1e-8, dK=TRUE)
da <- darg(list(mle=TRUE, max=0.5), designLHD(,rep(0,dim), rep(1,dim), control=list(size=1000)))
mleGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
ei2 <- plgpEI(gpi=gpi, x=start, fmin=ymin)
deleteGPsep(gpi)

```

plot.spotSeverity *Plot method for spotSeverity*

Description

Plot method for spotSeverity

Usage

```

## S3 method for class 'spotSeverity'
plot(
  x,
  add = FALSE,
  rangeLeft = -1,
  rangeRight = 1,
  plotSev = TRUE,

```

```

    plotPow = FALSE,
    cl = "black",
    xlab = "x",
    ylab = "y",
    ...
  )

```

Arguments

x	severity object
add	default value is FALSE
rangeLeft	range default:-1
rangeRight	range default:1
plotSev	logical. plot severity. Default: TRUE
plotPow	logical. plot power. Default: FALSE
cl	color, e.g., c("black", "red", "green", "blue", "brown", "cyan", "darkred", "gray", "green", "magenta", "orange")
xlab	x axis label
ylab	y axis label
...	additional parameters

Value

description of return value

Examples

```

### Example from D G Mayo and A Spanos.
### Severe Testing as a Basic Concept in a NeymanPearson Philosophy of Induction.
### British Journal for the Philosophy of Science, 57:323357, 2006. (fig 2):
x0 <- 12.1
mu1 <- seq(11.9,13,0.01)
n <- 100
sigma <- 2
alpha <- 0.025
tdist <- FALSE
plot(mu1, spotSeverity(xbar=x0, mu0=0, mu1=mu1, n=n, sigma=sigma, alpha=alpha,
tdist=tdist)$severity, type = "l", ylim=c(0,1), col="blue")
abline(h=0)
abline(h=1)
  abline(h=0.95)
abline(v=12.43)
### plot power:
mu0 <- 12
points(mu1, spotPower(alpha, mu0, mu1, n, sigma), type = "l", ylim=c(0,1),
col="green")
abline(v=12.72)

```

```
## Fig 13.11 in Span19a
p <- spotSeverity(xbar=10, mu0=10, mu1= 10.2, n=100, sigma = 1, alpha = 0.05, tdist = FALSE)
plot(p, rangeLeft = 10, rangeRight = 10.5, plotPow = TRUE)
```

plotBestObj	<i>Plot Best Objective Value</i>
-------------	----------------------------------

Description

Plot Best Objective Value

Usage

```
plotBestObj(y, end = length(y))
```

Arguments

y	result vector
end	length. Default: length(y)

Value

plot

plotData	<i>Interpolated plot</i>
----------	--------------------------

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	<i>Surface plot of a function</i>
--------------	-----------------------------------

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 $z=f(x_2, x_1)$ where Z is a two column matrix containing the sample locations x_1 and x_2 .
lower	boundary for x_1 and x_2 (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 = if (ncol(object$x) > 1 & tolower(type) != "singledim") {
    1:2
  } else {
    1
  },
  constant = object$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)
```

plotPCA

plotPCA

Description

plotPCA returns a 2D plot of optimization data in it's own space using buildPCA. It plots first two PCAs by default.

Usage

```
plotPCA(x, control = list())
```

Arguments

x	dataset of parameters to be transformed & plotted
control	control list

Value

It returns a plot image.

Author(s)

Alpar Gür <alpar.guer@smail.th-koeln.de>

See Also

[buildPCA](#), [biplot](#)

Examples

```

# define objective function
funGauss <- function (x) {
  gauss <- function(par) {
    y <- c(0.0009, 0.0044, 0.0175, 0.0540, 0.1295, 0.2420, 0.3521, 0.3989,
          0.3521, 0.2420, 0.1295, 0.0540, 0.0175, 0.0044, 0.0009)
    m <- 15
    x1 <- par[1]
    x2 <- par[2]
    x3 <- par[3]

    fsum <- 0
    for (i in 1:m) {
      ti <- (8 - i) * 0.5
      f <- x1 * exp(-0.5 * x2 * (ti - x3) ^ 2) - y[i]
      fsum <- fsum + f * f
    }
    return(fsum)
  }
  matrix(apply(x, # matrix
              1, # margin (apply over rows)
              gauss),
        , 1) # number of columns
}

# define starting point
x1 <- matrix(c(1,1,1),1,)
funGauss(x1)

# define boundaries
lower = c(-0.001,-0.007,-0.003)
upper = c(0.5,1.0,1.1)

res <- spot(funGauss, lower=lower, upper=upper, control=list(funEvals=15))

control = list(scale=TRUE) #pca control list, # scale the variables

plotPCA(res$x, control=control) # plot first two PCAs

```

plotPCAVariance

plotPCAVariance

Description

plotPCAVariance illustrates the total variance within the dataset. It plots the effectiveness of each principal component and can be used to decide how many and which principal components to plot. In order to create this plot, users don't need to build PCA beforehand since it handles this process automatically.

Usage

```
plotPCAVariance(x)
```

Arguments

x dataset of parameters to be transformed & plotted

Value

It returns a plot image.

Author(s)

Alpar Gür <alpar.guer@smail.th-koeln.de>

See Also

[buildPCA](#)

Examples

```
# objective function
funBard <- function (x) {
  bard <- function(par) {
    y <- c(0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58,
          0.73, 0.96, 1.34, 2.10, 4.39)
    m <- 15
    x1 <- par[1]
    x2 <- par[2]
    x3 <- par[3]

    fsum <- 0
    for (u in 1:m) {
      v <- 16 - u
      w <- min(u, v)
      f <- y[u] - (x1 + u / (v * x2 + w * x3))
      fsum <- fsum + f * f
    }
    return(fsum)
  }
  matrix(apply(x, # matrix
              1, # margin (apply over rows)
              bard),
        , 1) # number of columns
}

# starting point
x1 <- matrix(c(1,1),1,)
funBard(x1)

#boundaries
lower = c(-0.001,-0.007,-0.003)
```

```

upper = c(0.5,1.0,1.1)

res <- spot(funBard, lower=lower, upper=upper, control=list(funEvals=15))

plotPCAvariance(res$x) # plot variance within the dataset

```

predict.cvModel	<i>predict.cvModel</i>
-----------------	------------------------

Description

Predict with the cross validated model produced by [buildCVModel](#).

Usage

```

## S3 method for class 'cvModel'
predict(object, newdata, ...)

```

Arguments

object	CV model (settings and parameters) of class cvModel.
newdata	design matrix to be predicted
...	Additional parameters passed to the model

Value

prediction results: list with predicted mean ('y'), estimated uncertainty ('y'), linearly adapted uncertainty ('sLinear')

predict.spotBOModel	<i>Prediction method for bayesian optimization model</i>
---------------------	--

Description

Wrapper for predict.spotBOModel.

Usage

```

## S3 method for class 'spotBOModel'
predict(object, newdata, ...)

```

Arguments

object	fit of the model, an object of class "spotBOModel", produced by buildBO .
newdata	matrix of new data.
...	not used

Value

list with predicted mean y , uncertainty / standard deviation s (optional) and expected improvement ei (optional). Whether s and ei are returned is specified by the vector of strings `object$target`, which then contains "s" and "ei".

```
prepareBestObjectiveVal
```

Preprocess y Values to Plot Best Objective Value

Description

Preprocess y Values to Plot Best Objective Value

Usage

```
prepareBestObjectiveVal(y, end = length(y))
```

Arguments

<code>y</code>	result vector
<code>end</code>	length. Default: <code>length(y)</code>

Value

`prog`

```
repeatsOCBA
```

Optimal Computing Budget Allocation

Description

Simple interface to the Optimal Computing Budget Allocation algorithm.

Usage

```
repeatsOCBA(x, y, budget, verbosity = 0)
```

Arguments

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

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)
```

resBench01

result from the vignette benchmark

Description

A data set The corresponding code can be found in the vignette SPOTVignetteNutshell.

Usage

```
resBench01
```

Format

A list of

```
xbest num [1, 1:100] 188 45
```

 resSpot

S-Ring Simulation Data Obtained With SPOT

Description

A data set based on evaluations of the funCosts function. Second experiment (extension of the first design) The corresponding code can be found in the vignette SPOTVignetteElevator.

Usage

```
resSpot
```

Format

A list of 7:

xbest num [1, 1:2] 188 45

ybest num [1, 1] 1e+07

x num [1:87, 1:2] 17.4 143.6 89.9 28.7 51.4 ...

y num [1:87, 1] 1e+07 1e+07 1e+07 1e+07 1e+07 ...

count num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 ...

msg chr "budget exhausted"

modelFit List of 32

 resSpot2

S-Ring Simulation Data Obtained With SPOT

Description

A data set based on evaluations of the funCosts function. Second experiment (extension of the second design) The corresponding code can be found in the vignette SPOTVignetteElevator.

Usage

```
resSpot2
```

Format

A list of 7:

xbest num [1, 1:2] 188 45

ybest num [1, 1] 1e+07

x num [1:87, 1:2] 17.4 143.6 89.9 28.7 51.4 ...

y num [1:87, 1] 1e+07 1e+07 1e+07 1e+07 1e+07 ...

count num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 ...

msg chr "budget exhausted"

modelFit List of 32

ring

ring

Description

main function which iterates the ring

Usage

ring(params)

Arguments

params	list of
	randomSeed random seed
	nStates number of S-Ring states
	nElevators number of elevators
	probNewCustomer probability pf a customer arrival
	counter Counter: number of waiting customers
	sElevator Vector representing elevators (s)
	sCustomer Vector representing customers (c)
	currentState Current state that is calculated
	nextState Next state that is calculated
	nWeights Number of weights for the perceptron (= 2 * nStates)

Value

number of waiting customers (estimation)

runOptim	<i>runOptim</i>
----------	-----------------

Description

Run `optim` on a list of spot benchmark functions

Usage

```
runOptim(
  fl = makeMoreFunList(),
  method = "Nelder-Mead",
  n = 2,
  k = 1:length(makeMoreFunList())$funList,
  verbosity = 0
)
```

Arguments

<code>fl</code>	function list. Generated with one of the function list generators in <code>spot</code> , e.g., <code>makeSpotFunList</code> or <code>makeMoreFunList</code> . Default: <code>makeMoreFunList</code> .
<code>method</code>	The method used by <code>optim</code> : "Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", or "Brent". Default: "Nelder-Mead".
<code>n</code>	repeats. If <code>n > 1</code> , different start points (randomized) will be used. Default: <code>n=2</code> .
<code>k</code>	subset of benchmark functions. Default: <code>1:length(makeMoreFunList())\$funList</code> , i.e., all implemented functions.
<code>verbosity</code>	Level 0 shows no output (default).

Value

res. data.frame with results: `c("f", "r", "y")`

Examples

```
summary(runOptim(k=1)$y)
summary(runOptim(k=1, method="CG")$y)
```

runSpotBench	<i>runSpotBench</i>
--------------	---------------------

Description

Run [spot](#) on a list of spot benchmark functions

Usage

```
runSpotBench(  
  fl = makeMoreFunList(),  
  control = list(),  
  n = 2,  
  k = 1:length(makeMoreFunList())$funList),  
  verbosity = 0  
)
```

Arguments

fl	function list. Generated with one of the function list generators in spot , e.g., makeSpotFunList or makeMoreFunList . Default: makeMoreFunList .
control	The control list used by spot .
n	repeats. If $n > 1$, different start points (randomized) will be used. Default: $n=2$.
k	subset of benchmark functions. Default: $1:length(\text{makeMoreFunList}())\$funList$, i.e., all implemented functions.
verbosity	Level 0 shows no output (default).

Value

res. data.frame with results: `c("f", "r", "y")`

Examples

```
summary(runSpotBench(k=1)$y)
```

sann2spot *Interface SANN to SPOT*

Description

Provide an interface for tuning SANN. The interface function receives a `matrix` where each row is proposed parameter setting ('temp', 'tmax'), and each column specifies the parameters. It generates a $(n,1)$ -matrix as output, where n is the number of ('temp', 'tmax') parameter settings.

Usage

```
sann2spot(algpar, par = c(10, 10), fn, maxit = 100, ...)
```

Arguments

<code>algpar</code>	matrix algorithm parameters.
<code>par</code>	Initial values for the parameters to be optimized over.
<code>fn</code>	A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result.
<code>maxit</code>	Total number of function evaluations: there is no other stopping criterion. Defaults to 10000.
<code>...</code>	further arguments for <code>optim</code>

Value

matrix of results (performance values)

Examples

```
sphere <- function(x){sum(x^2)}
algpar <- matrix(c(1:10, 1:10), 10,2)
sann2spot(algpar, fn = sphere)
```

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)
```

simulate.kriging	<i>Kriging Simulation</i>
------------------	---------------------------

Description

(Conditional) Simulation at given locations, with a model fit resulting from `buildKriging`. In contrast to prediction or estimation, the goal is to reproduce the covariance structure, rather than the data itself. Note, that the conditional simulation also reproduces the training data, but has a two times larger error than the Kriging predictor.

Usage

```
## S3 method for class 'kriging'
simulate(
  object,
  nsim = 1,
  seed = NA,
  xsim,
  method = "decompose",
  conditionalSimulation = TRUE,
  Ncos = 10,
  returnAll = FALSE,
  ...
)
```

Arguments

object	fit of the Kriging model (settings and parameters), of class kriging.
nsim	number of simulations
seed	random number generator seed. Defaults to NA, in which case no seed is set
xsim	list of samples in input space, to be simulated at
method	"decompose" (default) or "spectral", specifying the method used for simulation. Note that "decompose" is can be preferable, since it is exact but may be computationally infeasible for high-dimensional xsim. On the other hand, "spectral" yields a function that can be evaluated at arbitrary sample locations.
conditionalSimulation	logical, if set to TRUE (default), the simulation is conditioned with the training data of the Kriging model. Else, the simulation is non-conditional.
Ncos	number of cosine functions (used with method="spectral" only)
returnAll	if set to TRUE, a list with the simulated values (y) and the corresponding covariance matrix (covar) of the simulated samples is returned.
...	further arguments, not used

Value

Returned value depends on the setting of object\$simulationReturnAll

References

N. A. Cressie. Statistics for Spatial Data. JOHN WILEY & SONS INC, 1993.

C. Lantuejoul. Geostatistical Simulation - Models and Algorithms. Springer-Verlag Berlin Heidelberg, 2002.

See Also

[buildKriging](#), [predict.kriging](#)

simulateFunction	<i>simulateFunction</i>
------------------	-------------------------

Description

Simulation-based Function Generator. Generate functions via simulation of Kriging models, e.g., for assessment of optimization algorithms with non-conditional or conditional simulation, based on real-world data.

Usage

```
simulateFunction(  
  object,  
  nsim = 1,  
  seed = NA,  
  method = "spectral",  
  xsim = NA,  
  Ncos = 10,  
  conditionalSimulation = TRUE  
)
```

Arguments

object	an object generated by buildKriging
nsim	the number of simulations, or test functions, to be created
seed	a random number generator seed. Defaults to NA; which means no seed is set. For sake of reproducibility, set this to some integer value.
method	"decompose" (default) or "spectral", specifying the method used for simulation. Note that "decompose" is can be preferable, since it is exact but may be computationally infeasible for high-dimensional xsim. On the other hand, "spectral" yields a function that can be evaluated at arbitrary sample locations.
xsim	list of samples in input space, for simulation (only used for decomposition-based simulation, not for spectral method)
Ncos	number of cosine functions (used with method="spectral" only)
conditionalSimulation	whether (TRUE) or not (FALSE) to use conditional simulation

Value

a list of functions, where each function is the interpolation of one simulation realization. The length of the list depends on the nsim parameter.

References

N. A. Cressie. Statistics for Spatial Data. JOHN WILEY & SONS INC, 1993.

C. Lantuejoul. Geostatistical Simulation - Models and Algorithms. Springer-Verlag Berlin Heidelberg, 2002.

See Also

[buildKriging](#), [simulate.kriging](#)

spot

*spot***Description**

Sequential Parameter Optimization. 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

<code>x</code>	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.
<code>fun</code>	is the objective function. It should receive a matrix <code>x</code> and return a matrix <code>y</code> . In case the function uses external code and is noisy, an additional seed parameter may be used, see the <code>control\$seedFun</code> argument below for details. Mostly, <code>fun</code> must have format <code>y = f(x, ...)</code> . If a noisy function requires some specific seed handling, e.g., in some other non-R code, a seed can be passed to <code>fun</code> . For that purpose, the user must specify <code>control\$noise = TRUE</code> and <code>fun</code> should be <code>fun(x, seed, ...)</code>
<code>lower</code>	is a vector that defines the lower boundary of search space. This determines also the dimensionality of the problem.
<code>upper</code>	is a vector that defines the upper boundary of search space.
<code>control</code>	is a list with control settings for <code>spot</code> . See spotControl .
<code>...</code>	additional parameters passed to <code>fun</code> .

Value

This function returns a list with:

<code>xbest</code>	Parameters of the best found solution (matrix).
<code>ybest</code>	Objective function value of the best found solution (matrix).
<code>x</code>	Archive of all evaluation parameters (matrix).
<code>y</code>	Archive of the respective objective function values (matrix).
<code>count</code>	Number of performed objective function evaluations.
<code>msg</code>	Message specifying the reason of termination.
<code>modelFit</code>	The fit of the last build model, i.e., an object returned by the last call to the function specified by <code>control\$model</code> .

Examples

```
## Only a few examples. More examples can be found in the vignette and in
## the paper "In a Nutshell -- The Sequential Parameter Optimization Toolbox",
## see https://arxiv.org/abs/1712.04076
```

```
## 1. Most simple example: Kriging + LHS search + predicted mean optimization
## (not expected improvement)
```

```
set.seed(1)
res <- spot(x=NULL, funSphere, c(-2, -3), c(1, 2),
           control=list(funEvals=15))
```

```
res$xbest
res$ybest
```

```
## 2. With expected improvement
```

```
set.seed(1)
res <- spot(x=NULL, funSphere, c(-2, -3), c(1, 2),
           control=list(funEvals=15,
                       modelControl=list(target="ei")))
```

```
res$xbest
res$ybest
```

```
### 3. Use local optimization instead of LHS search
```

```
set.seed(1)
res <- spot(, funSphere, c(-2, -3), c(1, 2),
           control=list(funEvals=15,
                       modelControl=list(target="ei"),
                       optimizer=optimLBFGSB))
```

```
res$xbest
res$ybest
```

```
### 4. Use transformed input values
```

```
set.seed(1)
f2 <- function(x){2^x}
lower <- c(-100, -100)
upper <- c(100, 100)
transformFun <- rep("f2", length(lower))
res <- spot(x=NULL, funSphere, lower=lower, upper=upper,
           control=list(funEvals=15,
                       modelControl=list(target="ei"),
                       optimizer=optimLBFGSB,
                       transformFun=transformFun))
```

```
res$xbest
res$ybest
```

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

<code>mue</code>	number of parents, default is 10
<code>nu</code>	selection pressure. That means, number of offspring (λ) is <code>mue</code> multiplied with <code>nu</code> . Default is 10
<code>dimension</code>	dimension number of the target function, default is 2
<code>mutation</code>	mutation type, either 1 or 2, default is 1
<code>sigmaInit</code>	initial sigma value (step size), default is 1.0
<code>nSigma</code>	number of different sigmas, default is 1
<code>tau0</code>	number, default is 0.0. <code>tau0</code> is the general multiplier.
<code>tau</code>	number, learning parameter for self adaption, default is 1.0. <code>tau</code> 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

spotCleanup

Clean up

Description

Remove objects

Usage

```
spotCleanup(control)
```

Arguments

control list of spot control parameters.

 spotControl

spotControl

Description

Default Control list for spot. This function returns the default controls for the functions [spot](#) and [spotLoop](#).

Usage

```
spotControl(dimension = NA)
```

Arguments

dimension problem dimension, that is, the number of optimized parameters. This parameter is mandatory since v2.8.4.

Details

Control is a list of the settings:

design A function that creates an initial design of experiment. Functions that accept the same parameters, and return a matrix like [designLHD](#) or [designUniformRandom](#) can be used. Default is [designLHD](#).

designControl A list of controls passed to the control list of the design function. See help of the respective function for details. Default is an empty list.

directOpt A function that is used to optimize after the spot run is finished. Functions that accept the same parameters, and return a matrix like [optimNLOPTR](#) or [optimDE](#) can be used. Default is [optimNLOPTR](#).

directOptControl A list of controls, which determine whether a direct optimization (exploitation of the final search region) is performed. Default is to run no direct optimization, i.e., `directOptControl = list(funEvals = 0)list`.

funEvals This is the budget of function evaluations of the direct optimization performed after the SMBO is performed. Default is `list(funEvals = 0)`.

duplicate In case of a deterministic (non-noisy) objective function, this handles duplicated candidate solutions. By default (`duplicate = "EXPLORE"`), duplicates are replaced by new candidate solutions, generated by random sampling with uniform distribution. If desired, the user can set this to "STOP", which means that the optimization stops and results are returned to the user (with a warning). This may be desirable, as duplicates can be a indicator for convergence, or for a problem with the configuration. In case of noise, duplicates are allowed.

funEvals This is the budget of function evaluations (spot uses no more than funEvals evaluations of fun), defaults to 20.

handleNAsMethod A function that treats NAs if there are any present in the result vector of the objective function. Default: NULL. By default NAs will not be treated.

- infillCriterion** A function defining an `infillCriterion` to be used while optimizing a model. Default: `NULL`. For example check `infillExpectedImprovement`
- model** A function that builds a statistical model of the observed data. Functions that accept the same parameters, and return a matrix like `buildKriging` or `buildRandomForest` can be used. Default is `buildKriging`.
- modelControl** A list of controls passed to the control list of the model function. See help of the respective function for details. Default is an empty list.
- multiStart** Number of restarts for optimization on the surrogate model. Default: 1, i.e., no restarts.
- noise** Boolean, whether the objective function has noise or not. Default is non-noisy, that is, `FALSE`.
- OCBA** Boolean, indicating whether Optimal Computing Budget Allocation (OCBA) should be used in case of a noisy objective function or not. OCBA controls the number of replications for each candidate solution. Note, that replicates should be larger than one in that case, and that the initial experimental design (see `design`) should also have replicates larger one. Default is `FALSE`.
- OCBABudget** The number of objective function evaluations that OCBA can distribute in each iteration. Default is 3.
- optimizer** A function that is used to optimize based on model, finding the most promising candidate solutions. Functions that accept the same parameters, and return a matrix like `optimLHD` or `optimDE` can be used. Default is `optimLHD`.
- optimizerControl** A list of controls passed to the control list of the optimizer function. See help of the respective function for details. Default is an empty list.
- parNames** Vector of parameter names of each variable as a string, defaults `c("x1", "x2", "x3", ...)`.
- plots** Whether progress should be tracked by a line plot, default is `FALSE`
- progress** Whether progress should be visualized, default is `FALSE`
- replicates** The number of times a candidate solution is initially evaluated, that is, in the initial design, or when created by the optimizer. Default is 1.
- replicateResult** logical. If `TRUE`, one result is replicated. The result is specified as the lower vector and re-evaluated `funEvals` times. No model building and optimization is performed, only evaluations on the objective function. Default: `FALSE`.
- returnFullControlList** logical. Return the full control list. Can be switched off to save memory/space. Default: `TRUE`.
- seedFun** An initial seed for the objective function in case of noise, by default `NA`. The default means that no seed is set. The user should be very careful with this setting. It is intended to generate reproducible experiments for each objective function evaluation, e.g., when tuning non-deterministic algorithms. If the objective function uses a constant number of random number generations, this may be undesirable. Note, that this seed is by default set prior to each evaluation. A replicated evaluation will receive an incremented value of the seed. Sometimes, the user may want to call external code using random numbers. To allow for that case, the user can specify an objective function (`fun`), which has a second parameter `seed`, in addition to first parameter (matrix `x`). This seed can then be passed to the external code, for random number generator initialization. See end of examples section for a demonstration.

- seedSPOT** This value is used to initialize the random number generator. It ensures that experiments are reproducible. Default is 1.
- subsetSelect** A function that selects a subset from a given set of design points. Default is [selectAll](#).
- subsetControl** A list of controls passed to the control list of the subsetSelect function. See help of the respective function for details. Default is an empty list.
- time** List with the following time information:
- maxTime** num Maximum allowed run time (in minutes) for spot or spotLoop. The default value for maxTime (in minutes) is Inf and can be overwritten by the user. The internal value **startTime**, that is used to control maxTime, will be set by [spotFillControllist](#). Note: maxTime is only an approximate value. It does not affect the directOpt run.
 - startTime** Start time. Will be set in [spotFillControllist](#).
 - endTime** End time.
- types** Vector of data type of each variable as a string, defaults "numeric" for all variables.
- verbosity** Integer level specifying how much output should be given by SPOT. 0 (default) ignores warnings of internal optimizers /models. 1 will show warnings and output.

Value

a list

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$funEvals`) by using `spotLoop`. This is the main loop of SPOT iterations. It requires the user to give the same inputs as specified for [spot](#). Note: `control$funEvals` must be larger than the value used in the previous run, because it specifies the total number of function evaluations and not the additional number of evaluations.

Usage

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

Arguments

- x** (m,n) matrix that contains the known candidate solutions. The SPOT loop is started with these values. Each row represents one n dimensional data point. Each of the m columns represents one optimized parameter.
- y** (m,p) matrix that represents observations for each point in x, Each of the m rows represents solutions for one data point.

fun	function that represents 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 <code>control\$seedFun</code> argument below for details.
lower	is a vector that defines the lower boundary of search space. This determines also the dimension 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.
## 5 additional runs will be performed.
control$funEvals <- 25
res2 <- spotLoop(res$x, res$y, funSphere, c(-2, -3), c(1, 2), control)
res2$xbest
res2$ybest
```

spotPlotErrors

spotPlotErrors

Description

Visualize the alpha, beta errors and the power of the test

Usage

```
spotPlotErrors(  
  alternative = "greater",  
  lower = -3,  
  upper = 3,  
  mu0 = 0,  
  mu1 = 1,  
  sigma = 1,  
  n = NULL,  
  xbar = 0,  
  alpha = 0.05,  
  beta = NULL  
)
```

Arguments

alternative	One of greater, less, or two.sided. The greater is the default.
lower	lower limit of the plot
upper	upper limit of the plot
mu0	mean of the null
mu1	mean of the alternative. See also parameter beta.
sigma	standard deviation
n	sample size
xbar	observed mean
alpha	error of the first kind
beta	error 2nd kind. Default NULL. If specified, then parameter mu1 will be ignored and mu1 will be calculated based on beta.

Value

description of return value

Examples

```
spotPlotErrors(lower=490,upper=510,mu0=500,mu1=504,sigma=2.7,n=9,xbar=502.22)  
spotPlotErrors(lower=140,upper=155,mu0=150,mu1=148,sigma=10,n=100,xbar=149,alternative="less")
```

spotPlotPower *spotPlotPower*

Description

Plot power

Usage

```
spotPlotPower(y0, y1, alpha = 0.05, add = FALSE, n = NA, rightLimit = 1)
```

Arguments

<code>y0</code>	First input vector
<code>y1</code>	Second input vector
<code>alpha</code>	description of alpha, default value is 0.05
<code>add</code>	Boolean, default value is FALSE
<code>n</code>	number of vector elements that should be evaluated, default value is NA, which means the whole vector
<code>rightLimit</code>	description of rightLimit, default value is 1

Value

description of return value

spotPlotSeverityBasic *spotPlotSeverityBasic*

Description

spotPlotSeverityBasic

Usage

```
spotPlotSeverityBasic(y0, y1, add = FALSE, n = NA, alpha, rightLimit = 1)
```

Arguments

<code>y0</code>	first input vector
<code>y1</code>	second input vector
<code>add</code>	default value is FALSE
<code>n</code>	default value is NA, which means length of y0 will be used for n
<code>alpha</code>	description
<code>rightLimit</code>	description of rightLimit, default value is 1

Value

description of return value

Examples

```
### Example from D G Mayo and A Spanos.
### Severe Testing as a Basic Concept in a NeymanPearson Philosophy of Induction.
### British Journal for the Philosophy of Science, 57:323357, 2006. (fig 2):
x0 <- 12.1
mu1 <- seq(11.9,13,0.01)
n <- 100
sigma <- 2
alpha <- 0.025
plot(mu1, spotSeverityBasic(x0, mu1, n, sigma, alpha), type = "l", ylim=c(0,1), col="blue")
abline(h=0)
abline(h=1)
  abline(h=0.95)
abline(v=12.43)
### plot power:
mu0 <- 12
points(mu1, spotPower(alpha, mu0, mu1, n, sigma), type = "l", ylim=c(0,1), col="green")
abline(v=12.72)
```

spotPlotTest

spotPlotTest

Description

Visualize test result, errors, and severity

Usage

```
spotPlotTest(
  alternative = "greater",
  lower = -3,
  upper = 3,
  mu0 = 0,
  mu1 = 1,
  sigma = 1,
  n = NULL,
  xbar = 0,
  alpha = 0.05,
  beta = NULL
)
```


Arguments

alternative	One of greater, less, or two.sided. Full plots are currently implemented for less, which is the default.
lower	lower limit of the plot
upper	upper limit of the plot
mu0	mean of the null
mu1	mean of the alternative. See also parameter beta.
sigma	standard deviation
n	sample size
xbar	observed mean
alpha	error of the first kind
beta	error 2nd kind. Default NULL. If specified, then parameter mu1 will be ignored and mu1 will be calculated based on beta.

Value

description of return value

Examples

```
spotPlotTest(lower=490, upper=510, mu0=500, mu1=504, sigma=2.7, n=9, xbar=502.22, alpha=0.025)
## The following two plots should be nearly identical:
spotPlotTest(lower=490, upper=510, mu0=500, sigma=2.7, n=9, xbar=502.22, alpha=0.025, beta=0.2)
spotPlotTest(lower=490, upper=510, mu0=500, mu1=502.5215, sigma=2.7, n=9, xbar=502.22, alpha=0.025)
```

spotPower

spotPower

Description

Calculate power

Usage

```
spotPower(alpha, mu0, mu1, n, sigma)
```

Arguments

alpha	description of alpha
mu0	description of mu0
mu1	description of mu1
n	vector length
sigma	standart deviation

Value

description of return value

spotSeverity	<i>spotSeverity</i>
--------------	---------------------

Description

spotSeverity

Usage

```
spotSeverity(xbar, mu0, mu1, n, sigma, alpha, tdist = FALSE, paired = TRUE)
```

Arguments

xbar	sample mean value
mu0	mean value of the null hypothesis (usually referred to as H0)
mu1	mean value of the alternative hypothesis (usually referred to as H1)
n	sample size in each arm, e.g., if 20 samples are available, then n=10 regardless whether the samples are paired/blocked (paired=TRUE) or independent (paired=FALSE). Degrees of freedom will be modified internally according to the setting of the paired argument.
sigma	sample s.d. Will be used to determine s.d. of the differences (if paired==TRUE) or s.d. of the pooled s.d (if paired==FALSE).
alpha	probability of a type I error, given H0 is true
tdist	logical. Use Student t Distribution. Default: FALSE
paired	logical. Paired (blocked) data. Default: TRUE

Value

an object of class "spotSeverity", with a summary method and a print method.

Examples

```
s0 <- spotSeverity(xbar=0.4, mu0=0.0, mu1=0.6, n=25, sigma=1, alpha=0.03)
print(s0)
s1 <- spotSeverity(xbar=0.4, mu0=0.6, mu1=0.6, n=25, sigma=1, alpha=0.03)
print(s1)
s2 <- spotSeverity(xbar=0, mu0=0.6, mu1=0.6, n=25, sigma=1, alpha=0.03)
print(s2)
```

```
## Example from Mayo, p345
spotSeverity(xbar=90, mu0=0, mu1= 200, n=200, sigma = 450, alpha = 0.025,
paired = FALSE, tdist = FALSE)
```

```
## Example from Vena02a to compare with results from t.test()
## library("BHH2")
## data(shoes.data)
## A <- shoes.data$matA
## B <- shoes.data$matB
A <- c(13.2, 8.2, 10.9, 14.3, 10.7, 6.6, 9.5, 10.8, 8.8, 13.3)
B <- c(14, 8.8, 11.2, 14.2, 11.8, 6.4, 9.8, 11.3, 9.3, 13.6)
t.paired <- t.test(x = A, y = B, var.equal = TRUE, paired = TRUE,
alternative = "greater", conf.level = 0.95)
xbar <- mean(A-B)
n <- length(A)
sigma <- sd(A-B)
s.paired <- spotSeverity(xbar=xbar,mu0=0, mu1= 1, n=n, sigma = sigma,
alpha = 0.025, tdist = TRUE)
```

spotSeverityBasic *spotSeverityBasic*

Description

spotSeverityBasic

Usage

spotSeverityBasic(x0, mu1, n, sigma, alpha)

Arguments

x0	sample mean value
mu1	description
n	description
sigma	description
alpha	description

Value

description of return value

sring	<i>sring</i>
-------	--------------

Description

simple elevator simulator

Usage

```
sring(x, opt = list(), ...)
```

Arguments

x	perceptron weights
opt	list of optional parameters, e.g., nElevators number of elevators probNewCustomer probability pf a customer arrival nIterations Number of iterations randomSeed random seed
...	additional parameters

Value

fitness

Examples

```
set.seed(123)
nStates = 6
nElevators = 2
sigma = 1
x = matrix( rnorm(n = 2*nStates, 1, sigma), 1, )
sring(x, opt = list(nElevators=nElevators,
                  nStates= nStates) )
```

sringRes1	<i>S-Ring Simulation Data</i>
-----------	-------------------------------

Description

A data set based on evaluations of the funCosts function. The corresponding code can be found in the vignette SPOTVignetteElevator

Usage

```
sringRes1
```

Format

A data frame with 20 obs. of 3 variables:

```
y num 10 10 10 10 10 ...
```

```
sigma num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 ..
```

```
ne num 5 5 5 5 5 5 5 5 5 ...
```

```
sringRes2
```

```
S-Ring Simulation Data
```

Description

A data set based on evaluations of the `funCosts` function. Second experiment (extension of the first design) The corresponding code can be found in the vignette `SPOTVignetteElevator`

Usage

```
sringRes2
```

Format

A data frame with 22 obs. of 3 variables:

```
y num 10 10 10 10 10 ...
```

```
sigma num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 ..
```

```
ne num 5 5 5 5 5 5 5 5 5 ...
```

```
sringRes3
```

```
S-Ring Simulation Data
```

Description

A data set based on evaluations of the `funCosts` function. Second experiment (extension of the first design) The corresponding code can be found in the vignette `SPOTVignetteElevator`

Usage

```
sringRes3
```

Format

A data frame with 27 obs. of 3 variables:

y num 1e+07 1e+07 1e+07 1e+07 1e+07 ...

sigma num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 ...

ne num 5 5 5 5 5 5 5 5 5 ...

thetaNugget	<i>thetaNugget</i>
-------------	--------------------

Description

get theta (distance, lengthscale) and nugget (noise) parameters gradient

Usage

thetaNugget(par, X, Y)

Arguments

par parameter vector. First dim(x) entries are theta values, last entry is nugget parameter.

X x coordinates

Y y values at x

Value

negLogLikelihood

thetaNuggetGradient	<i>thetaNuggetGradient</i>
---------------------	----------------------------

Description

get theta (distance, lengthscale) and nugget (noise) parameters gradient

Usage

thetaNuggetGradient(par, X, Y)

Arguments

par parameter vector. First dim(x) entries are theta values, last entry is nugget parameter.

X x coordinates

Y y values at x

transformX	<i>Transform input</i>
------------	------------------------

Description

Transform input variables

Usage

```
transformX(xNat = NA, fn = vector())
```

Arguments

xNat	matrix with natural variables. Default: NA.
fn	vector of transformation functions names (char). Default: Empty vector (vector()).

Value

matrix of transformed parameters

Examples

```
f2 <- function(x){2^x}
fn <- c("identity", "exp", "f2")
xNat <- diag(3)
transformX(xNat, fn)

fn <- append(fn, c("sin", "cos", "tan"))
xNat <- cbind(xNat, xNat)
transformX(xNat, fn)
```

vmessage	<i>formatted output dependent on verbosity</i>
----------	--

Description

Combine [sprintf](#) and [writeLines](#) to generate formatted output

Usage

```
vmessage(verbosity, text, value)
```

Arguments

verbosity	verbosity level
text	output to be printed
value	value to be printed

Examples

```
x <- 123
vmessage(1, "value of x:" , x)
```

wrapBatchTools	<i>wrapBatchTools</i>
----------------	-----------------------

Description

Wrap a given objective function to be evaluated via the batchtools package and make it accessible for SPOT.

Usage

```
wrapBatchTools(  
  fun,  
  reg = NULL,  
  clusterFunction = batchtools::makeClusterFunctionsInteractive(),  
  resources = NULL  
)
```

Arguments

fun	function to wrap
reg	batchtools registry, if none is provided, then one will be created automatically
clusterFunction	batchtools clusterFunction, default: makeClusterFunctionsInteractive()
resources	resource list that is passed to batchtools, default NULL

Value

callable function for SPOT

wrapFunction	<i>Function Evaluation Wrapper</i>
--------------	------------------------------------

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))
```

wrapFunctionParallel *Parallelized 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](#). While doing so, the wrapper will use the parallel package in order to parallelize the execution of each function evaluation. This function will create a computation cluster if no cluster is specified and there is no default cluster setup!

Usage

```
wrapFunctionParallel(fun, cl = NULL, nCores = NULL)
```

Arguments

fun	the function that shall be evaluated in parallel
cl	Optional, an existing computation cluster
nCores	Optional, amount of cores to use for creating a new computation cluster. Default is all cores.

Value

numeric vector, result of the parallelized evaluation

wrapSystemCommand *wrapSystemCommand*

Description

Optimize parameters for a script that is accessible via Command Line

Usage

```
wrapSystemCommand(systemCall)
```

Arguments

systemCall	String that calls the command line script.
------------	--

Value

callable function for SPOT

Examples

```
# exampleScriptLocation <- system.file("consoleCallTrialScript.R",package = "SPOT")
# f <- wrapSystemCommand(paste("${R_HOME}/bin/Rscript", exampleScriptLocation))
# spot(,f,c(1,1),c(100,100))
```

Index

* datasets

dataGasSensor, 25
resBench01, 93
resSpot, 94
resSpot2, 94
springRes1, 116
springRes2, 117
springRes3, 117

* package

SPOT-package, 5

* spotTools

diff0, 29

biplot, 88

buildBO, 5, 91

buildCVModel, 7, 91

buildEnsembleStack, 8

buildGaussianProcess, 9

buildKriging, 10, 66, 67, 99–101, 107

buildKrigingDACE, 13

buildLasso, 14

buildLM, 15

buildLOESS, 16

buildPCA, 17, 88, 90

buildRandomForest, 18, 87, 107

buildRanger, 19

buildrsdummy, 21

buildRSM, 21, 26

buildTreeModel, 22

checkArrival, 23

checkFeasibilityNlopGnIngres, 24

code2nat, 24

corrCubic, 13

correxP, 13

correxPg, 13

corrGauss, 13

corrKriging, 13

corrLin, 13

corrNoisyGauss, 13

corrNoisyKriging, 13

corrSpherical, 13

corrSpline, 13

dataGasSensor, 25

descentSpotRSM, 26

designLHD, 27, 75, 77, 106

designUniformRandom, 28, 106

diff0, 29

doParallel, 30

expectedImprovement, 30

funBard, 31

funBeale, 32

funBox3d, 32

funBranin, 33

funBrownBs, 34

funCosts, 35

funCyclone, 35

funError, 37

funFreudRoth, 38

funGauss, 39

funGoldsteinPrice, 39

funGulf, 40

funHelical, 41

funIshigami, 42

funJennSamp, 43

funMeyer, 44

funMoo, 45

funNoise, 45

funOptimLecture, 46

funPowellBs, 47

funPowellS, 47

funRosen, 48

funRosen2, 49

funShiftedSphere, 50, 52

funSoblev99, 50

funSphere, 37, 50, 51, 105

funSpring, 52

- getCosts, 53
- getMultiStartPoints, 54
- getNatDesignFromCoded, 54
- getPerformanceStats, 55
- getPosition, 55
- getPower, 56
- getReplicates, 57
- getSampleSize, 57

- handleNAsKrigingWorst, 58
- handleNAsMax, 59
- handleNAsMean, 60

- imputeY, 61
- infillEI, 62
- infillExpectedImprovement, 62
- init_ring, 63
- is.finite, 37

- makeMoreFunList, 64, 96, 97
- makeSpotFunList, 65, 96, 97

- normalizeMatrix, 66
- normalizeMatrix2, 67

- obj.plgpEI, 67
- objectiveFunctionEvaluation, 68
- OCBA, 93
- ocbaRanking, 70
- optim, 65, 96
- optimDE, 71, 106, 107
- optimES, 72, 104
- optimGenoud, 74
- optimLagp, 75
- optimLBFGB, 76
- optimLHD, 77, 107
- optimNLOPTR, 78, 106
- optimRSfun, 79

- perceptron, 80
- plgpEI, 68, 80
- plot.spotSeverity, 81
- plotBestObj, 83
- plotData, 83, 86, 88
- plotFunction, 84, 85, 88
- plotModel, 84, 86, 87
- plotPCA, 17, 88
- plotPCAVariance, 89
- predict.cvModel, 91
- predict.dace, 14

- predict.ensembleStack, 9
- predict.kriging, 6, 11–13, 100
- predict.spotBOModel, 7, 91
- predict.spotLOESS, 17
- predict.spotRSM, 22
- prepareBestObjectiveVal, 92

- regpoly0, 13
- regpoly1, 13
- regpoly2, 13
- repeatsOCBA, 71, 92
- resBench01, 93
- resSpot, 94
- resSpot2, 94
- ring, 95
- runOptim, 96
- runSpotBench, 97

- sann2spot, 98
- satter, 98
- selectAll, 108
- simulate.kriging, 99, 101
- simulateFunction, 100
- SPOT (SPOT-package), 5
- spot, 5, 14, 24, 54, 61, 68, 69, 97, 102, 106, 108, 121, 122
- SPOT-package, 5
- spotAlgEs, 103
- spotCleanup, 105
- spotControl, 24, 54, 55, 59–61, 69, 71, 102, 106, 109
- spotFillControllist, 108
- spotLoop, 24, 54, 106, 108
- spotPlotErrors, 109
- spotPlotPower, 111
- spotPlotSeverityBasic, 111
- spotPlotTest, 112
- spotPower, 113
- spotSeverity, 114
- spotSeverityBasic, 115
- sprintf, 119
- sring, 52, 116
- sringRes1, 116
- sringRes2, 117
- sringRes3, 117

- thetaNugget, 118
- thetaNuggetGradient, 118
- transformX, 68, 69, 119

`vmessage`, [119](#)

`wrapBatchTools`, [120](#)

`wrapFunction`, [121](#)

`wrapFunctionParallel`, [122](#)

`wrapSystemCommand`, [122](#)

`writeLines`, [119](#)