Package ‘SPOT’

March 18, 2022

License GPL (>= 2)
Title Sequential Parameter Optimization Toolbox
Type Package
LazyLoad yes
LazyData true
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 2.8.2
Date 2022-03-16
Depends R (>= 3.5.0)
Imports DEoptim, ggplot2, glmnet, graphics, grDevices, laGP, MASS, nloptr, plgp, plotly, rpart, randomForest, ranger, rgenoud, rsm, stats, utils
RoxygenNote 7.1.2
Suggests babsim.hospital, 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 Zaefferer [aut] (<https://orcid.org/0000-0003-2372-2092>),
  Frederik Rehbach [aut] (<https://orcid.org/0000-0003-0922-8629>),
  Margarita Rebolledo [ctb],

1
R topics documented:

SPOT-package .................................................. 4
buildBO .......................................................... 5
buildCVMModel ................................................. 7
buildEnsembleStack .......................................... 8
buildGaussianProcess ....................................... 9
buildKriging ................................................... 10
buildKrigingDACE ............................................ 12
buildLasso ...................................................... 14
buildLM .......................................................... 15
buildLOESS ..................................................... 16
buildPCA ........................................................ 17
buildRandomForest ......................................... 18
buildRanger .................................................... 19
buildrsdummy .................................................. 20
buildRSM ........................................................ 21
buildTreeModel ............................................... 22
checkArrival .................................................. 23
checkFeasibilityNlopGnIngres .............................. 23
code2nat ........................................................ 24
dataGasSensor .................................................. 24
descentSpotRSM ............................................... 26
designLHD ....................................................... 26
designUniformRandom ....................................... 28
diff0 ............................................................... 29
doParallel ...................................................... 29
expectedImprovement ....................................... 30
funBaBSimHospital ........................................... 30
funBard .......................................................... 32
funBeale ........................................................ 32
funBox3d ........................................................ 33
funBranin ....................................................... 34
funBrownBs .................................................... 35
funCosts ........................................................ 36
funCyclone ...................................................... 36
funFreudRoth .................................................. 38
funGauss ......................................................... 39
funGoldsteinPrice ........................................... 40
funGulf .......................................................... 40
funHelical ....................................................... 41
R topics documented:

funIshigami ................................................. 42
funJennSamp ........................................... 43
funMeyer ................................................... 44
funMoo ...................................................... 45
funOptimLecture ........................................ 45
funPowellBs ............................................. 46
funPowellS ............................................... 47
funRosen ................................................... 48
funRosen2 .................................................. 48
funShiftedSphere ....................................... 49
funSoblev99 ............................................... 50
funSphere .................................................. 51
funSring .................................................... 51
getCosts .................................................... 52
getMultiStartPoints .................................... 53
getNatDesignFromCoded .................................. 53
getPower .................................................... 54
getSampleSize ............................................ 55
infillEI ...................................................... 56
infillExpectedImprovement .............................. 57
init_ring ..................................................... 57
makeMoreFunList ........................................ 59
makeSpotFunList .......................................... 60
normalizeMatrix ........................................... 60
normalizeMatrix2 .......................................... 61
obj.plgpEI .................................................. 62
optimDE ..................................................... 63
optimES ..................................................... 64
optimGenoud .............................................. 66
optimLagp ................................................... 67
optimLBFGSB ............................................... 68
optimLHD ................................................... 69
optimNLOPTR .............................................. 70
optimRSfun .................................................. 71
perceptron .................................................. 72
plgpEI ....................................................... 72
plot.spotSeverity ......................................... 73
plotBestObj ................................................. 75
plotData .................................................... 75
plotFunction ............................................... 77
plotModel ................................................... 79
plotPCA .................................................... 80
plotPCAvariance .......................................... 81
predict.cvModel .......................................... 83
predict.spotBOModel ...................................... 83
prepareBestObjectiveVal .................................. 84
repeatsOCBA ............................................... 84
resBench01 .................................................. 85
## Description

Sequential Parameter Optimization Toolbox

## Details

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

buildBO

Bayesian Optimization Model Interface

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

See Also

Main interface function is spot.
**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**


**See Also**

`predict.spotBOModel`

**Examples**

```r
## 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 BO Model
m1 <- buildBO(x = matrix(x, ncol = 1),
y = matrix(y, ncol=1),
control = list(target="ei"))
```
buildCVModel

```r
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)))
```

---

**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**

```r
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, buildKriging)).
  - `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

See Also
predict.ensembleStack
Examples

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

---

### buildGaussianProcess

**Gaussian Process Model Interface**

---

**Description**

Gaussian Process Model Interface

**Usage**

```r
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**

```r
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)
c## Print model parameters
print(fit)
c## Predict at new location
xNew <- matrix( c(-0.1, 0.1), ncol = 1)
predict(fit, xNew)
c## 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

control  
(list), with the options for the model building procedure:

types a character vector giving the data type of each variable. All but "factor" will be handled as numeric, "factor" (categorical) variables will be subject to the hamming distance.

thetaLower lower boundary for theta, default is \text{1e-4}

thetaUpper upper boundary for theta, default is \text{1e2}

algTheta algorithm used to find theta, default is optimDE.

budgetAlgTheta budget for the above mentioned algorithm, default is \text{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 \text{-6}

lambdaUpper upper boundary for log10lambda, default is \text{0}

startTheta optional start value for theta optimization, default is NULL

reinterpolate whether (TRUE, default) or not (FALSE) reinterpolation should be performed target target values of the prediction, a vector of strings. Each string specifies a value to be predicted, e.g., "y" for mean, "s" for standard deviation, "ei" for expected improvement. See also predict.kriging. This can also be changed after the model has been built, by manipulating the respective object$target value.

Details

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

Value

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

References


See Also

predict.kriging

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 <- as.matrix(apply(x,1,braninFunction))
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))
```
**buildKrigingDACE**

---

**Build DACE model**

**Description**

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

**Usage**

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

**Arguments**

- `x` design matrix (sample locations), rows for each sample, columns for each variable.
**buildKrigingDACE**

`y` vector of observations at `x`

`control` (list), with the options for the model building procedure:

- `startTheta` optional start value for theta optimization, default is `NULL`
- `algTheta` algorithm used to find theta, default is `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.
- `nugget` Value for nugget. Default is -1, which means the nugget will be optimized during MLE. Else it can be fixed in a range between 0 and 1.
- `regr` Regression function to be used: `regpoly0` (default), `regpoly1`, `regpoly2`. Can be a custom user function.
- `corr` Correlation function to be used: `corrnoisykriging` (default), `corrkriging`, `corrnoisygauss`, `corrgauss`, `correxpg`, `correxpg`, `correnc`, `corrspherical`, `corrspline`. Can also be user supplied (if in the right form).
- `target` target values of the prediction, a vector of strings. Each string specifies a value to be predicted, e.g., "y" for mean, "s" for standard deviation, "ei" for expected improvement. See also `predict.kriging`. This can also be changed after the model has been build, by manipulating the respective object$target value.

**Value**

returns an object of class dace with the following elements:

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

**Author(s)**

The authors of the original DACE Matlab toolbox are Hans Bruun Nielsen, 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**


**See Also**

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

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) {
  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**

```r
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**

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

---

## buildLOESS

**Build LOESS Model**

### Description

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

### Usage

```r
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

```r
## Create a test function: branin
braninFunction <- function (x) {
  10 * (1 - 1/(8 * pi))^2 * 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

---

**Description**

`buildPCA` builds principal components of given dataset. It is used inside `plotPCA` function to build necessary object to perform principal components analysis.

**Usage**

```r
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@smaill.th-koeln.de>
Examples

# define objective function

```r
```

```r
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
```

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

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

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

---

**buildRandomForest**  
*Random Forest Interface*

**Description**

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

**Usage**

```r
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

```r
## Test-function:
braininFunction <- function (x) {
  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,braininFunction))
## Create model
fit <- buildRandomForest(x,y)
## Print model parameters
print(fit)
## Predict at new location
predict(fit,cbind(1,2))
## True value at location
braininFunction(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

- `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. These are all configuration parameters of the ranger function, and will be passed on to it.

Value

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

```r
## 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**  
*Build random search dummy model*

### 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

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

### Arguments

- `x`  
  x (independent variables), not used.
- `y`  
  y (dependent variable), not used.
- `control`  
  control, not used.
Description
Using the rsm package, this function builds a linear response surface model.

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

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

Value
returns an object of class spotRSM.

See Also
predict.spotRSM

Examples
## Create a test function: branin
braninFunction <- function (x) {
  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
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

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 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)
```r
fit <- buildTreeModel(x, y)
# plot(fit)
# plot(x, y, type = "l")
yhat <- predict(fit, newdata = 1)
yhat$y == 10
```

---

**checkArrival**

**Description**

Calculate arrival events for S-Ring.

**Usage**

```r
checkArrival(probNewCustomer)
```

**Arguments**

- `probNewCustomer` probability of an arrival of a new customer

**Value**

logical

**Examples**

```r
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**

```r
cHECKFEASIBILITYNLOPNLOPNINGS(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

Gas Sensor Data

Description

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

Usage

dataGasSensor
Format

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

Y  Measured Sensor Output
X1 Sensor Input 1
X2 Sensor Input 2
X3 Sensor Input 3
X4 Sensor Input 4
X5 Sensor Input 5
X6 Sensor Input 6
X7 Sensor Input 7
Batch Disturbance variable, measurement batch
Sensor Disturbance variable, sensor ID
Set1 test/training split, 1 is training data, 2 is test data
Set2 test/training split
Set3 test/training split
Set4 test/training split
Set5 test/training split
Set6 test/training split
Set7 test/training split
Set8 test/training split
Set9 test/training split
Set10 test/training split

Details

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

References

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

*Descent RSM model*

**Description**

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

**Usage**

```r
descentSpotRSM(object)
```

**Arguments**

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

**Value**

list with

- `x` list of points along the path of steepest descent
- `y` corresponding predicted values

**See Also**

`buildRSM`

---

### designLHD

*Latin Hypercube Design Generator*

**Description**

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

**Usage**

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

- **x**: optional matrix x, rows for points, columns for dimensions. This can contain one or more points which are part of the design, but specified by the user. These points are added to the design, and are taken into account when calculating the pair-wise distances. They do not count for the design size. E.g., if x has two rows, control$replicates is one and control$size is ten, the returned design will have 12 points (12 rows). The first two rows will be identical to x. Only the remaining ten rows are guaranteed to be a valid LHD.

- **lower**: vector with lower boundary of the design variables (in case of categorical parameters, please map the respective factor to a set of contiguous integers, e.g., with lower = 1 and upper = number of levels)

- **upper**: vector with upper boundary of the design variables (in case of categorical parameters, please map the respective factor to a set of contiguous integers, e.g., with lower = 1 and upper = number of levels)

- **control**: list of controls:
  - **size**: number of design points
  - **retries**: number of retries during design creation
  - **types**: this specifies the data type for each design parameter, as a vector of either "numeric", "integer", "factor". (here, this only affects rounding)
  - **inequalityConstraint**: inequality constraint function, smaller zero for infeasible points. Used to replace infeasible points with random points.
  - **replicates**: integer for replications of each design point. E.g., if replications is two, every design point will occur twice in the resulting matrix.

**Value**

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

**Author(s)**

Original code by Christian Lasarczyk, adaptations by Martin Zaefferer

**Examples**

```r
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**

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

**Arguments**

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

**Value**

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

**Examples**

```r
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)
```
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

Parallel execution of code, dependent on the operating system

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  Expected Improvement**

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

**Usage**
expectedImprovement(mean, sd, min)

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

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

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

**funBaBSimHospital  Optimization of the BaBSim.Hospital Simulator**

**Description**
funBaBSimHospital implements an interface to the babsim.hospital package. babsim.hospital is a discrete-event simulation model for a hospital resource planning problem. The project is motivated by the challenges faced by health care institutions in the COVID-19 pandemic. It can be used by health departments to forecast demand for intensive care beds, ventilators, and staff resources. funBaBSimHospital provides an interface to getTrainTestObjFun.
funBaBSimHospital

Usage

funBaBSimHospital(
  x,
  region = 5374,
  nCores = 2,
  verbosity = 0,
  rkiEndDate = "2020-12-09",
  icuEndDate = "2020-12-09",
  trainingWeeksSimulator = 10,
  trainingWeeksField = 6,
  totalRepeats = 10
)

Arguments

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

region integer. Represents the region code. Default: 5374 (Oberberg).
nCores integer. Defines the number of cores.
verbosity integer. Handles output. Default: 0
rkiEndDate characters. Last day of rki data. Default "2020-12-09"
icuEndDate characters. Last day of icu data. Default "2020-12-09"
trainingWeeksSimulator integer. Training period using rki data. Default: 10. Should be larger than trainingWeeksField.
totalRepeats integer. Number of repeats for each configuration. Should be a multiple of nCores. Default: 10.

Value

y numeric function value.

Examples

## babsim.hospital version must be greater equal 11.7:
# ver <- unlist(packageVersion("babsim.hospital"))
#   x <- matrix(as.numeric(babsim.hospital::getParaSet(5374)[1,-1]),1,)
#   funBaBSimHospital(x)
# }
funBard (No. 14, More No. 8)

Description

3-dim Bard Test Function

\[ x_0 = (1,1,1) \]
\[ f = 8.21487\ldots 10^{-3} \]
\[ f = 17.4286\ldots \text{ at } (0.8406\ldots, -\infty, -\infty) \]

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


Examples

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

funBeale (No.11, More No. 5)

Description

2-dim Beale Test Function

Usage

funBeale(x)
funBox3d

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


Examples

```r
x1 <- matrix(c(1,1),1,)
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
funBranin

References


@examples

```r
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
plotModel(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**

```r
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**

```r
x1 <- matrix(c(-pi, 12.275),1,)
funBranin(x1)
```
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 x0 = (1,1) f=0 at (1e6, 2e-6)

Value

1-column matrix with resulting function values

References


Examples

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

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].
funCyclone

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

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

Value

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

References


funFreudRoth

funFreudRoth (No. 8, More No. 2)

Description

2-dim Freudenstein 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


funGauss

Examples

```r
x1 <- matrix(c(1,1),1,)
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 (No. 15, More No. 9)

Description

3-dim Gaussian Test Function

Usage

```r
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

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

**Goldstein-Price Test Function (No. 5)**

**Description**

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

**Usage**

```r
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**

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

---

funGulf  

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

**Description**

3-dim Gulf research and development Test Function

**Usage**

```r
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
funHelical

Value

1-column matrix with resulting function values

References


Examples

```r
x1 <- matrix(c(50,25,1.5),1,)
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
funIshigami

References

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_3^4sin(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 ~ 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 >= -pi and <= 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
funJennSamp

References


Examples

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

---

funJennSamp (No. 12, More No 6)

Description

2-dim Jennrich and Sampson Function Test Function

Usage

```r
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


Examples

```r
x1 <- matrix(c(1,1),1,)
funJennSamp(x1)
res1 <- spot(x,funJennSamp,c(0,0),c(0.3,0.3))
plotModel(res1$model)
```
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


Examples

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

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 )

funOptimLecture

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

Usage
funOptimLecture(vec)

Arguments
vec input vector or matrix of candidate solution
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


Examples

x1 <- matrix(c(-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

Powells 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


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

ware. ACM Transactions on Mathematical Software (TOMS), 7(1), 17-41. doi: 10.1145/355934.355936

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)
funShiftedSphere

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

```r
x1 <- matrix(c(-pi, 12.275),1,)
funRosen2(x1)
```
funSoblev99

Description

An implementation of the Sobol-Levitan function.

\[ f(x) = \exp(\text{sum } b_i x_i) - I_d + c_0, \text{ where } I_d = \prod (\exp(b_i) - 1) / b_i \]

The value of the elements in the b-vector \((b_1, ..., b_d)\) affect the importance of the corresponding \(x\)-variables. Sobol’ & Levitan (1999) use two different b-vectors: \((1.5, 0.9, 0.9, 0.9, 0.9, 0.9)\), for \(d = 6\), and \((0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4)\), for \(d = 20\). Our implementation uses the default b vector: \(b = c(0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4)\) (when \(d\leq20\)).

Moon et al. (2012) scale the output to have a variance of 100. For \(d = 20\), they use three different b-vectors: \((2, 1.95, 1.9, 1.85, 1.8, 1.75, 1.7, 1.65, 0.4228, 0.3077, 0.2169, 0.1471, 0.0951, 0.0577, 0.0323, 0.0161, 0.0068, 0.0021, 0.0004, 0)\), \((1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)\), 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 \(c0\) is \(c0 = 0\). The function is evaluated on \(xi\) in \([0,1]\), for all \(i = 1, ..., d\).

Usage

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

Arguments

- **x**: \((m,2)\)-matrix of points to evaluate with the function. Values should be \(\geq 0\) and \(\leq 1\), i.e., \(x_i\) in \([0,1]\).
- **b**: \(d\)-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 \(d\leq20\))
- **c0**: constant term (optional), with default value \(0\)

Value

1-column matrix with resulting function values

References


Examples

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

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

Description

wrapper for sring

Usage

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

funShiftedSphere
Arguments

x         perceptron weights
opt       list of optional parameters, e.g.,
nElevators number of elevators
probNewCustomer probability of customer arrival
nIterations Number of iterations
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

Description

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

Examples

set.seed(123)
sigma = 1
ten = 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 \pm 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)
getPower

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

```
# Note: donttest is used, because platform x86_64-w64-mingw32 (64-bit)
# does not provide the package babsim.hospital.

require(babsim.hospital)
x <- matrix(rep(-1,29),1,)
bounds <- getBounds()
lower <- bounds$lower
upper <- bounds$upper
getNatDesignFromCoded(x, a = lower, b=upper)
```

getPower

Description

Implements basic power calculations in R See also: [https://www.cyclismo.org/tutorial/R/power.html](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".
getSampleSize

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

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

mu0  mean value of the null hypothesis (usually referred to as H0)
mu1  mean value of the alternative hypothesis (usually referred to as H1)
alpaha  type I error
beta  type II error
sigma  sample s.d.
alternative  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")

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 can be passed to control$infillCriterion in order to be used during the optimization in SPOT. Parameters dont have to be specified as this function is meant 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

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$\text{modelControl}\text{infillCriterion}$ in order to be used during the optimization in SPOT. Parameters don't have to be specified as this function is meant 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

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

init_ring

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)
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 \code{optim}.

Value

list of functions with starting points and optimum points.

References


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

**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**

**Description**
Normalize design by using minimum and maximum of the design values for input space. Supportive function for Kriging model, not to be used directly.

**Usage**
normalizeMatrix(x, ymin, ymax)

**Arguments**
- x: design matrix in input space
- ymin: minimum vector of normalized space
- ymax: maximum vector of normalized space
**normalizeMatrix2**

**Value**

normalized design matrix

**See Also**

buildKriging

---

**normalizeMatrix2**

*Normalize design 2*

**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
Wrapper for Expected improvement (Gramacy)

Usage

```r
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

```r
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)
```
optimDE

deleteGPsep(gpi)

---

**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**

```r
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

```r
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**

```r
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


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

Minimization by L-BFGS-B

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

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

**Examples**

```r
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**

```r
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**

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

**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**

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

**Arguments**

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

Note that the arguments `x`, `fun`, `lower` and `upper` will be mapped to the corresponding arguments of nloptr: `x0`, `eval_f`, `lb` and `ub`.

**Value**

list, with elements

- `x` archive of evaluated solutions
- `y` archive of observations
- `xbest` best solution
- `ybest` best observation
- `count` number of evaluations of `fun`
- `message` success message
**Examples**

```r
##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**

```r
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

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 2xnStates, for each state (sElevator/sCustomer) there is one input

Value
logical pass or take decision

plgpEI

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

```r
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)))
mlGPsep(gpi, param="d", tmin=da$min, tmax=da$max, ab=da$ab)
ei2 <- plgpEI(gpi=gpi, x=start, fmin=ymin)
deleteGPsep(gpi)
```

Description

Plot method for spotSeverity

Usage

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

```r
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**

```r
### 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

Plot Best Objective Value

Description
Plot Best Objective Value

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

Arguments

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

Value
plot

plotData

Interpolated plot

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

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

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.

ey  
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

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

plotFunction

Surface plot of a function

Description

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

Usage

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

Arguments

f function to be plotted. The function should either be able to take two vectors or one matrix specifying sample locations. i.e. \( z = f(X) \) or \( z = f(x2, x1) \) where \( Z \) is a two column matrix containing the sample locations \( x1 \) and \( x2 \).

lower boundary for \( x1 \) and \( x2 \) (defaults to \( c(0, 0) \)).

upper boundary (defaults to \( c(1, 1) \)).

type string describing the type of the plot: "filled.contour" (default), "contour", "persp" (perspective), or "persp3d" plot. Note that "persp3d" is based on the plotly package and will work in RStudio, but not in the standard RGui.
The `plotFunction` function allows you to plot functions in R with various customization options. Here are some of the key arguments and their descriptions:

- `s`: number of samples along each dimension. e.g. \( f \) will be evaluated \( s^2 \) times.
- `xlab`: label of first axis
- `ylab`: label of second axis
- `zlab`: label of third axis
- `color.palette`: colors used, default is `terrain.color`.
- `title`: 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**

```r
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

```r
## 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)
```

Description

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

Usage

```r
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

```r
# 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
```

---

### 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)
predict.cvModel

```r
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
```

---

**Description**

Predict with the cross validated model produced by `buildCVModel`.

**Usage**

```r
## 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**

*Prediction method for bayesian optimization model*

**Description**

Wrapper for `predict.spotBOModel`.

**Usage**

```r
## 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 \( e_i \) (optional). Whether \( s \) and \( e_i \) 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**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>result vector</td>
</tr>
<tr>
<td>( \text{end} )</td>
<td>length. Default: ( \text{length}(y) )</td>
</tr>
</tbody>
</table>

---

**repeatsOCBA**

*Optimal Computing Budget Allocation*

**Description**

A simple interface to the Optimal Computing Budget Allocation algorithm.

**Usage**

`repeatsOCBA(x, y, budget)`

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>matrix of samples. Identical rows indicate repeated evaluations. Any sample should be evaluated at least twice, to get an estimate of the variance.</td>
</tr>
<tr>
<td>( y )</td>
<td>observations of the respective samples. For repeated evaluations, ( y ) should differ (variance not zero).</td>
</tr>
<tr>
<td>( \text{budget} )</td>
<td>of additional evaluations to be allocated to the samples.</td>
</tr>
</tbody>
</table>
Value

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

References


See Also

repeatsOCBA calls OCBA, which also provides some additional details.

Examples

```r
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
**ring**

**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 1e+07 ...
- `count` num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 1 ...
- `msg` chr "budget exhausted"
- `modelFit` List of 32

---

**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 of 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

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

fl function list. Generated with one of the function list generators in spot, e.g., makeSpotFunList or makeMoreFunList. Default: makeMoreFunList.

method The method used by optim: "Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN", or "Brent". Default: "Nelder-Mead".

n repeats. If n > 1, different start points (randomized) will be used. Default: n=2.

k subset of benchmark functions. Default: 1:length(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(runOptim(k=1)$y)
summary(runOptim(k=1, method="CG")$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**

\[
sann2spot(\text{algpar}, \text{par} = c(10, 10), \text{fn}, \text{maxit} = 100, \ldots)
\]

**Arguments**

- **algpar** matrix algorithm parameters.
- **par** Initial values for the parameters to be optimized over.
- **fn** 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.
- **maxit** Total number of function evaluations: there is no other stopping criterion. Defaults to 10000.
- **...** further arguments for \texttt{optim}

**Value**

matrix of results (performance values)

**Examples**

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

**satter**  

**Satterthwaite Function**

**Description**

The Satterthwaite function can be used to estimate the magnitude of the variance component $(\text{sigma}_\beta)^2$, when the random factor has significant main effects.

**Usage**

\[
satter(\text{MScoeff}, \text{MSi}, \text{dfi}, \text{alpha} = 0.05)
\]
simulate.kriging

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MScoeff</td>
<td>coefficients c_1, c_2</td>
</tr>
<tr>
<td>MSi</td>
<td>mean squared values</td>
</tr>
<tr>
<td>dfi</td>
<td>degrees of freedom</td>
</tr>
<tr>
<td>alpha</td>
<td>error probability</td>
</tr>
</tbody>
</table>

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

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

simulate.kriging  

Kriging Simulation

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

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

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


See Also

buildKriging, predict.kriging

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


See Also

buildKriging, simulate.kriging
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

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

Value

This function returns a list with:

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

## 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(x=NULL, 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

mue number of parents, default is 10
nu selection pressure. That means, number of offspring (lambda) is mue multiplied with nu. Default is 10
dimension dimension number of the target function, default is 2
mutation mutation type, either 1 or 2, default is 1
sigmaInit initial sigma value (step size), default is 1.0
nSigma number of different sigmas, default is 1
tau0 number, default is 0.0. tau0 is the general multiplier.
tau number, learning parameter for self adaption, default is 1.0. tau is the local multiplier for step sizes (for each dimension).
rho  number of parents involved in the procreation of an offspring (mixing number), default is "bi"

sel  number of selected individuals, default is -1


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  

**Description**

Remove objects

**Usage**

spotCleanup(control)

**Arguments**

control  list of spot control parameters.
Description

Default Control list for spot. This function returns the default controls for the functions spot and spotLoop.

Usage

spotControl(dimension)

Arguments

dimension problem dimension, that is, the number of optimized parameters.

Details

Control is a list of the settings:

funEvals This is the budget of function evaluations (spot uses no more than funEvals evaluations of fun), defaults to 20.
multiStart Number of restarts for optimization on the surrogate model. Default: 1, i.e., no restarts.
types Vector of data type of each variable as a string, defaults "numeric" for all variables.
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.
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.
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.
opimizer 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.
opimizerControl 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.
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).

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

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.

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.

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.

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.

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.

plots Whether progress should be tracked by a line plot, default is FALSE

progress Whether progress should be visualized, default is FALSE
infillCriterion A function defining an infillCriterion to be used while optimizing a model. Default: NULL. For example check infillExpectedImprovement

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.

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.

fixError logical. If eval returns an error, replace return value with mean(already evaluated values). Default: FALSE.

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**

```r
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 control$seedFun 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

Description

Plot power

Usage

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

- `y0`: First input vector
- `y1`: Second input vector
- `alpha`: Description of `alpha`, default value is 0.05
- `add`: Boolean, default value is FALSE
- `n`: Number of vector elements that should be evaluated, default value is NA, which means the whole vector
- `rightLimit`: Description of `rightLimit`, default value is 1

Value

description of return value

---

Description

`spotPlotSeverityBasic`

Usage

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

Arguments

- `y0`: first input vector
- `y1`: second input vector
- `add`: default value is FALSE
- `n`: default value is NA, which means length of `y0` will be used for `n`
- `alpha`: description
- `rightLimit`: 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):
\[
x_0 \leftarrow 12.1 \\
m_1 \leftarrow \text{seq}(11.9,13,0.01) \\
n \leftarrow 100 \\
sigma \leftarrow 2 \\
alpha \leftarrow 0.025
\]

plot(m1, spotSeverityBasic(x0, m1, 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:
\[
mu0 \leftarrow 12 \\
points(m1, spotPower(alpha, mu0, m1, n, sigma), type = "l", ylim=c(0,1), col="green")
abline(v=12.72)
\]

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alternative</td>
<td>One of greater, less, or two.sided. Full plots are currently implemented for less, which is the default.</td>
</tr>
<tr>
<td>lower</td>
<td>lower limit of the plot</td>
</tr>
<tr>
<td>upper</td>
<td>upper limit of the plot</td>
</tr>
</tbody>
</table>
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

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
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/blocke(d (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)

---

### 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
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 of 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) )

S-Ring Simulation Data

Description

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

sringRes2

Format

A data frame with 20 obs. of 3 variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>num 10 10 10 10 ...</td>
</tr>
<tr>
<td>sigma</td>
<td>num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 ...</td>
</tr>
<tr>
<td>ne</td>
<td>num 5 5 5 5 5 5 5 5 5 5 ...</td>
</tr>
</tbody>
</table>

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

sringRes3

Format

A data frame with 22 obs. of 3 variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>num 10 10 10 10 ...</td>
</tr>
<tr>
<td>sigma</td>
<td>num 0.1 0.1 0.1 0.1 0.1 1 1 1 1 1 ...</td>
</tr>
<tr>
<td>ne</td>
<td>num 5 5 5 5 5 5 5 5 5 5 ...</td>
</tr>
</tbody>
</table>

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
thetaNuggetGradient

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 5 ...

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

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  

Transform input

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)

wrapBatchTools

wrapBatchTools

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
)

wrapFunction

### 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

---

**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

```r
### example function
branin <- function (x) {
  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))
```
**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**

**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

```r
# 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, 24
  resBench01, 85
  resSpot, 86
  resSpot2, 86
  sringRes1, 106
  sringRes2, 107
  sringRes3, 107
* package
  SPOT-package, 4
* spotTools
  diff0, 29

biplot, 80
buildBO, 5, 83
buildCVModel, 7, 83
buildEnsembleStack, 8
buildGaussianProcess, 9
buildKriging, 10, 61, 90–92, 97
buildKrigingDACE, 12
buildLasso, 14
buildLM, 15
buildLOESS, 16
buildPCA, 17, 80, 82
buildRandomForest, 18, 79, 97
buildRanger, 19
buildrsdummy, 20
buildRSM, 21, 26
buildTreeModel, 22

checkArrival, 23
checkFeasibilityNlopGnIngres, 23
code2nat, 24
corrcubic, 13
correxp, 13
correxpg, 13
corrgauss, 13
corrrkriqing, 13
corrlin, 13
corrrnoisygauss, 13
corrrnoisykriging, 13
corrspherical, 13
corrspline, 13
dataGasSensor, 24
descentSpotRSM, 26
designLHD, 26, 67, 69, 97
designUniformRandom, 28, 97
diff0, 29
doParallel, 29
expectedImprovement, 30
funBABSImHospital, 30
funBard, 32
funBeale, 32
funBox3d, 33
funBrannin, 34
funBrownBs, 35
funCosts, 36
funCyclone, 36
funFreudRoth, 38
funGauss, 39
funGoldsteinPrice, 40
funGulf, 40
funHelical, 41
funIshigami, 42
funJenSamp, 43
funMeyer, 44
funMoo, 45
funOptimLecture, 45
funPowellBs, 46
funPowellS, 47
funRosen, 48
funRosen2, 48
funShiftedSphere, 49, 51
funSoblev99, 50
funSphere, 49, 51, 96
funSring, 51
getCosts, 52
getMultiStartPoints, 53
getNatDesignFromCoded, 53
getPower, 54
getSampleSize, 55
getTrainTestObjFun, 30

infillEI, 56
infillExpectedImprovement, 57
init_ring, 57

makeMoreFunList, 59, 88
makeSpotFunList, 60, 88

normalizeMatrix, 60
normalizeMatrix2, 61

obj.plgpEI, 62
OCBA, 85

optim, 59, 60, 88
optimDE, 63, 97, 98
optimES, 64, 95
optimGenoud, 66
optimLagp, 67
optimLBFGSB, 68
optimLHD, 69, 97
optimNL0PTR, 70, 98
optimR5fun, 71

perceptron, 72
plgpEI, 62, 72
plot.spotSeverity, 73
plotBestObj, 75
plotData, 75, 78, 79
plotFunction, 76, 77, 79
plotModel, 76, 78, 79
plotPCA, 17, 80
plotPCAvariance, 81
predict.cvModel, 83
predict.dace, 13
predict.ensembleStack, 8
predict.kriging, 5, 10, 11, 13, 91
predict.spotBOModel, 6, 83
predict.spotLOESS, 16
predict.spotRSM, 21
prepareBestObjectiveVal, 84

regpoly0, 13
regpoly1, 13
regpoly2, 13
repeatsOCBA, 84

resBench01, 85
resSpot, 86
resSpot2, 86
ring, 87
runOptim, 88
sann2spot, 89
satter, 89
selectAll, 97
simulate.kriging, 90, 92
simulateFunction, 91
SPOT (SPOT-package), 4
spot, 5, 14, 23, 53, 93, 97, 99, 110, 111
SPOT-package, 4
spotAlgEs, 94
spotCleanup, 96
spotControl, 23, 53, 93, 97, 100
spotFillControlList, 99
spotLoop, 23, 53, 97, 99
spotPlotPower, 100
spotPlotSeverityBasic, 101
spotPlotTest, 102
spotPower, 103
spotSeverity, 104
spotSeverityBasic, 105
sring, 51, 106
sringRes1, 106
sringRes2, 107
sringRes3, 107

thetaNugget, 108
thetaNuggetGradient, 108
transformX, 109

wrapBatchTools, 109
wrapFunction, 110
wrapFunctionParallel, 111
wrapSystemCommand, 111