Package ‘SPOT’

March 8, 2017

License GPL (>= 2)
Title Sequential Parameter Optimization Toolbox
Type Package
LazyLoad yes
LazyData true
Description A set of tools for model based optimization and tuning of algorithms. It includes surrogate models, optimizers and design of experiment approaches. The main interface is spot, which uses sequentially updated surrogate models for the purpose of efficient optimization. The main goal is to ease the burden of objective function evaluations, when a single evaluation requires a significant amount of resources.
Version 2.0.1
Date 2017-03-08
Depends R (>= 3.0.0)
Imports randomForest, stats, utils, graphics, MASS
RoxygenNote 6.0.1
Suggests testthat
NeedsCompilation no
Author Thomas Bartz-Beielstein [aut],
Joerg Stork [aut],
Martin Zaefferer [aut, cre],
Margarita Rebolledo [ctb],
Christian Lasarczyk [ctb],
Joerg Ziegenhirt [ctb],
Wolfgang Konen [ctb],
Oliver Flasch [ctb],
Patrick Koch [ctb],
Martina Friese [ctb]
Maintainer Martin Zaefferer <martin.zaefferer@gmx.de>
Repository CRAN
Date/Publication 2017-03-08 16:37:21
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.

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

As this is the initial re-written version, we would appreciate feedback about any bugs or other issues with the package. Feel free to send feedback by mail to the maintainer.

<table>
<thead>
<tr>
<th>Package:</th>
<th>SPOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Package</td>
</tr>
<tr>
<td>Version:</td>
<td>2.0.1</td>
</tr>
<tr>
<td>Date:</td>
<td>2017-03-08</td>
</tr>
<tr>
<td>License:</td>
<td>GPL (&gt;= 2)</td>
</tr>
<tr>
<td>LazyLoad:</td>
<td>yes</td>
</tr>
</tbody>
</table>

Acknowledgments

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

Maintainer

Martin Zaefferer <martin.zaefferer@gmx.de>

Author(s)

Thomas Bartz-Beielstein <thomas.bartz-beielstein@th-koeln.de>, Joerg Stork, Martin Zaefferer (Maintainer, <martin.zaefferer@gmx.de>) with contributions from: C. Lasarczyk, M. Rebolledo

See Also

Main interface function is `spot`.

---

**buildEnsembleStack**  
*Ensemble: Stacking*

**Description**

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

**Usage**

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

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
## Create a test function: bramin
braminFunction <- 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,braminFunction))
## Create model with default settings
fit <- buildEnsembleStack(x,y)
## Predict new point
predict(fit,cbind(1,2))
## True value at location
braminFunction(c(1,2))
```
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, reinterpolation is also by default turned on. The model uses a Gaussian kernel.

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

Arguments
- **x**: design matrix (sample locations)
- **y**: vector of observations at x
- **control**: (list), with the options for the model building procedure:
  - **types**: a character vector giving the data type of each variable. All but "factor" will be handled as numeric, "factor" (categorical) variables will be subject to the hamming distance.
  - **thetaLower**: lower boundary for theta, default is 1e-4
  - **thetaUpper**: upper boundary for theta, default is 1e2
  - **algTheta**: algorithm used to find theta, default is optimLBFGSB.
  - **budgetAlgTheta**: budget for the above mentioned algorithm, default is 200. The value will be multiplied with the length of the model parameter vector to be optimized.
  - **optimizep**: boolean that specifies whether the exponents (p) should be optimized. Else they will be set to two. Default is FALSE
  - **useLambda**: whether or not to use the regularization constant lambda (nugget effect). Default is TRUE.
  - **lambdaLower**: lower boundary for log10lambda, default is ~6
  - **lambdaUpper**: upper boundary for log10lambda, default is 0
  - **startTheta**: optional start value for theta optimization, default is NULL
  - **reinterpolate**: whether (TRUE, default) or not (FALSE) reinterpolation should be performed target target values of the prediction, a vector of strings. Each string specifies a value to be predicted, e.g., "y" for mean, "s" for standard deviation, "ei" for expected improvement. See also predict.kriging. This can also be changed after the model has been built, by manipulating the respective object$target value.

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)
buildKriging

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

## Test-function:
braininFunction <- function(x) {
  10 * (1 - 1/(8 * pi)) * cos(x[1]) + 10
}
## Create design points
set.seed(1)
x <- cbind(runif(15)*15-5,runif(15)*15)
## Compute observations at design points (for Branin function)
y <- as.matrix(apply(x,1,braininFunction))
## 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
braininFunction(c(1,2))
##
## Next Example: Handling factor variables

### create a test function:

```r
braninFunctionFactor <- function(x) {
  10 * (1 - 1/(8 * pi)) * cos(x[1]) + 10
  if(x[3]==1)
    y <- y +1
  else if(x[3]==2)
    y <- y -1
  y
}
```

### create training data

```r
set.seed(1)
x <- cbind(runif(50)*15-5,runif(50)*15,sample(1:3,50,replace=TRUE))
y <- as.matrix(apply(x,1,braninFunctionFactor))
```

### fit the model (default: assume all variables are numeric)

```r
fitDefault <- buildKriging(x,y,control = list(algTheta=optimLBFGSB))
```

### fit the model (give information about the factor variable)

```r
fitFactor <- buildKriging(x,y,control = list(algTheta=optimLBFGSB,types=c("numeric","numeric","factor")))
```

### create test data

```r
xtest <- cbind(runif(200)*15-5,runif(200)*15,sample(1:3,200,replace=TRUE))
ytest <- as.matrix(apply(xtest,1,braninFunctionFactor))
```

### Predict test data with both models, and compute error

```r
ypredDef <- predict(fitDefault,xtest)$y
ypredFact <- predict(fitFactor,xtest)$y
mean((ypredDef-ytest)^2)
mean((ypredFact-ytest)^2)
```

---

**buildKrigingDACE**  
**Build DACE model**

**Description**

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

**Usage**

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

**Arguments**

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

- `y`  
  vector of observations at x
control (list), with the options for the model building procedure:
  startTheta optional start value for theta optimization, default is NULL
  algTheta algorithm used to find theta, default is optimLBFGSB.
  budgetAlgTheta budget for the above mentioned algorithm, default is 200. The
  value will be multiplied with the length of the model parameter vector to be op-
  timized.
  nugget Value for nugget. Default is -1, which means the nugget will be op-
  timized 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), corrkrriging,
  corrnoisygauss, corrgauss, correxpg, correxpg, correcpp, corrcubic, corrspherical, corrspline.
  Can also be user supplied (if in the right form). target target values of the pre-
  diction, 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 http://www2.imm.dtu.dk/projects/dace/
are Hans Bruun Nielsen <hbn@imm.dtu.dk>, Soren Nymand Lophaven and Jacob Sondergaard.
Extension of the Matlab code by Tobias Wagner <wagner@isf.de>.
Porting and adaptation to R and further extensions by Martin Zaefferer <martin.zaefferer@fh-koeln.de>.

References

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

See Also

  predict.dace
**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 built 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
## 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)
```
Examples

```r
## 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 <- 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))
```

**Description**

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

**Usage**

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

**Arguments**

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

**Value**

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

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

---

**buildSO**  
*Second Order Linear Model Interface*

**Description**

This is a simple wrapper for the lm function, which fits second order 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 built with main effects and interactions as well as quadratic effects. Optionally, the model is also subject to the AIC-based stepwise algorithm, using the step function from the stats package.

**Usage**

`buildSO(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 "spotSecondOrderModel", with a predict method and a print method.
Examples

```r
## 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 <- buildSO(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))
```

---

dataGasSensor

**Gas Sensor Data**

**Description**

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

**Usage**

dataGasSensor

**Format**

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

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

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

References

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

---

**designLHD**  *Latin Hypercube Design Generator*

**Description**

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

**Usage**

\[
designLHD(x = \text{NULL}, \text{lower}, \text{upper}, \text{control} = \text{list()})
\]
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)
- replicates integer for replications of each design point. E.g., if replications is two, every design point will occur twice in the resulting matrix.

Value

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

Author(s)

Original code by Christian Lasarczyk, adaptations by Martin Zaefferer

Examples

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

Description

Create a simple experimental design based on uniform random sampling.

Usage

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

Arguments

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

Value

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

Examples

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

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

**Arguments**

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

**Value**

A vector with the negative logarithm of the expected improvement values, \(-\log_{10}(EI)\).

**Examples**

```r
generate a vector with the negative logarithm of the expected improvement values:

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)
```

---

**funCyclone**

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

**Description**

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

**Usage**

```r
funCyclone(x, deterministic = c(T, T, T), cyclone = list(Da = 1260, H = 2500, Dt = 420, Ht = 650, He = 600, Be = 200), fluid = list(Mu = 18.5 * 10^(-6), Vp = 5000, Rhop = 2000, Rhof = 1.204, Croh = 50), noiseLevel = list(Vp = 0.1, Rhop = 0.05))
```
Arguments

x         vector of length at least one and up to six, specifying non-default geometrical parameters: Da, H, Dt, Ht, He, Be
deterministic  binary vector. First element specifies whether volume flow is deterministic or not. Second element specifies whether particle density is deterministic or not. Third element specifies whether particle diameters are deterministic or not. Default: All are deterministic (TRUE).
cyclone   list of a default cyclone’s geometrical parameters: fluid$Da, fluid$H, fluid$Dt, fluid$Ht, fluid$He and fluid$Be
fluid     list of default fluid parameters: fluid$Mu, fluid$Vp, fluid$Rhop, fluid$Rhof and fluid$Croh
noiseLevel list of noise levels for volume flow (noiseLevel$Vp) and particle density (noiseLevel$Rhop), only used if non-deterministic.

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

References


Examples

```r
## Call directly
funcyclone(c(1260,2500))
## create vectorized target function, vectorized, first objective only
## Also: negated, since SPOT always does minimization.
tfunvecF1 <- function(x){-apply(x,1,funcyclone)[1,]}
tfunvecF1(matrix(c(1260,2500,1000,2000),2,2,byrow=TRUE))
## optimize with spot
res <- spot(fun=tfunvecF1,lower=c(1000,2000),upper=c(2000,3000),
            control=list(modelControl=list(target="ei"),
                          model=buildKriging,optimizer=optimLBFGS,plots=TRUE))
## best found solution ...
res$xbest
## ... and its objective function value
res$ybest
```
funSphere  

*Sphere Test Function*

**Description**
Sphere Test Function

**Usage**
funSphere(x)

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

**Value**
1-column matrix with resulting function values

**Examples**
funSphere(matrix(runif(18),3))

---

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  
only matrix of points. Only first point (row) is used as startpoint.
fun  
ojective 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 <- optimLHSB(fun = funSphere, lower = c(-10, -20), upper = c(20, 8))
res$ybest
```

Descripción

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.
  - `retries` Number of retries for design generation, used by `designLHD`
  - `...` passed to `fun`
Value

- list, with elements
- \(x\) archive of evaluated solutions
- \(y\) archive of observations
- \(x_{\text{best}}\) best solution
- \(y_{\text{best}}\) best observation
- count number of evaluations of \(\text{fun}\)
- message success message

Examples

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

---

**repeatsOCBA**  
*Optimal Computing Budget Allocation*

Description

A simple interface to the Optimal Computing Budget Allocation algorithm.

Usage

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

Arguments

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

Value

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

References


See Also

repeatsOCBA calls \texttt{OCBA}, which also provides some additional details.
Examples

\begin{verbatim}
x <- matrix(c(1:3,1:3),9,2)
y <- runif(9)
repeatsOCBA(x,y,10)
\end{verbatim}

Description

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

Usage

\begin{verbatim}
spot(x = NULL, fun, lower, upper, control = list(), ...)
\end{verbatim}

Arguments

\begin{itemize}
  \item \texttt{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.
  \item \texttt{fun} is the objective function. It should receive a matrix \texttt{x} and return a matrix \texttt{y}. In case the function uses external code and is noisy, an additional seed parameter may be used, see the \texttt{control$seedFun} argument below for details.
  \item \texttt{lower} is a vector that defines the lower boundary of search space. This determines also the dimensionality of the problem.
  \item \texttt{upper} is a vector that defines the upper boundary of search space.
  \item \texttt{control} is a list of additional settings:
    \begin{itemize}
      \item \texttt{funEvals} This is the budget of function evaluations (spot uses no more than \texttt{funEvals} evaluations of \texttt{fun}), defaults to 20.
      \item \texttt{types} Vector of data type of each variable as a string, defaults "numeric" for all variables.
      \item \texttt{design} A function that creates an initial design of experiment. Functions that accept the same parameters, and return a matrix like \texttt{designLHD} or \texttt{designUniformRandom} can be used. Default is \texttt{designLHD}.
      \item \texttt{designControl} A list of controls based to the \texttt{control} list of the design function. See help of the respective function for details. Default is an empty list.
      \item \texttt{model} A function that builds a statistical model of the observed data. Functions that accept the same parameters, and return a matrix like \texttt{buildKriging} or \texttt{buildRandomForest} can be used. Default is \texttt{buildKriging}.
    \end{itemize}
\end{itemize}
modelControl A list of controls based to the control list of the model function. See help of the respective function for details. Default is an empty list.

optimizer A function that is used to optimize based on model, finding the most promising candidate solutions. Functions that accept the same parameters, and return a matrix like optimLHD or optimLBFGB can be used. Default is optimLHD.

optimizerControl A list of controls based to the control list of the optimizer function. See help of the respective function for details. Default is an empty list.

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.

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.

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

... 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

```r
## Most simple example: Kriging + LHS + predicted
## mean optimization (not expected improvement)
res <- spot(funSphere,c(-2,-3),c(1,2))
res$xbest

## With expected improvement
res <- spot(funSphere,c(-2,-3),c(1,2),
            control=list(modelControl=list(target="ei")))
res$xbest

## With additional start point:
# res <- spot(matrix(c(0.05,0.1),1,2),funSphere,c(-2,-3),c(1,2))
# res$xbest
# res <- spot(funSphere,c(-2,-3),c(1,2),
#              control=list(funEvals=50))
# res$xbest

## Use a local optimizer instead of LHS
res <- spot(funSphere,c(-2,-3),c(1,2),
            control=list(optimizer=optimLBFGSB))
res$xbest

## Random Forest instead of Kriging
res <- spot(funSphere,c(-2,-3),c(1,2),
            control=list(model=buildRandomForest))
res$xbest

## LM instead of Kriging
res <- spot(funSphere,c(-2,-3),c(1,2),
            control=list(model=buildLM)) #lm as surrogate
res$xbest

## LM and local optimizer (which for this simple example is perfect)
res <- spot(funSphere,c(-2,-3),c(1,2),
            control=list(model=buildLM, optimizer=optimLBFGSB))
res$xbest

## Or a different Kriging model:
res <- spot(funSphere,c(-2,-3),c(1,2),
            control=list(model=buildKrigingDACE, optimizer=optimLBFGSB))
res$xbest

## With noise: (this takes some time)
res1 <- spot(function(x)funSphere(x)+rnorm(nrow(x)),c(-2,-3),c(1,2),
```
# control=list(funEvals=100,noise=TRUE)) # noisy objective
# res2 <- spot(function(x)funSphere(x)+rnorm(nrow(x)),c(-2,-3),c(1,2),
# control=list(funEvals=100,noise=TRUE,replicates=2,
# designControl=list(replicates=2))) # noise with replicated evaluations
# res3 <- spot(function(x)funSphere(x)+rnorm(nrow(x)),c(-2,-3),c(1,2),
# control=list(funEvals=100,noise=TRUE,replicates=2,OCBA=T,OCBABudget=1,
# designControl=list(replicates=2))) # and with OCBA
### Check results with non-noisy function:
# funSphere(res1$xbest)
# funSphere(res2$xbest)
# funSphere(res3$xbest)
### The following is for demonstration only, to be used for random number
### seed handling in case of external noisy target functions.
# res3 <- spot(function(x,seed)(set.seed(seed);funSphere(x)+rnorm(nrow(x))),
# c(-2,-3),c(1,2),control=list(funEvals=100,noise=TRUE,seedFun=1))
###
### Next example: Handling factor variables
### Note: factors should be coded as integer values, i.e., 1,2,...,n
### create a test function:
braninFunctionFactor <- function (x) {
       10 * (1 - 1/(8 * pi)) * cos(x[1]) + 10
  if(x[3]==1)
    y <- y +1
  else if(x[3]==2)
    y <- y -1
  y
}
### vectorize
objFun <- function(x,apply(x,1,braninFunctionFactor))
set.seed(1)
res <- spot(fun=objFun,lower=c(-5,0,1),upper=c(10,5,3),
            control=list(model=buildKriging,
                          types= c("numeric","numeric","factor"),
                          optimizer=optimLHD))
res$x$xbest
res$y$ybest
Index

*Topic datasets
  dataGasSensor, 12
*Topic package
  SPOT-package, 2
  buildEnsembleStack, 3
  buildKriging, 5, 21
  buildKrigingDACE, 7
  buildLM, 9
  buildRandomForest, 10, 21
  buildS0, 11
  corrcubic, 8
  correxp, 8
  correxpg, 8
  corrgauss, 8
  corrkriging, 8
  corrlin, 8
  corrnoisygauss, 8
  corrnoisykriging, 8
  corrspherical, 8
  corrspline, 8
  dataGasSensor, 12
  designLHD, 13, 19, 21
  designUniformRandom, 15, 21
  expectedImprovement, 16
  funCyclone, 16
  funSphere, 18

OCBA, 20
optimLBFGSB, 18, 22
optimLHD, 19, 22

predict.dace, 8
predict.ensembleStack, 4
predict.kriging, 5, 6, 8
regpoly0, 8

regpoly1, 8
regpoly2, 8
repeatsOCBA, 20

SPOT (SPOT-package), 2
spot, 3, 21
SPOT-package, 2