Package ‘SACOBRA’

March 26, 2020

Type Package
Title Self-Adjusting COBRA
Version 1.2
Date 2020-03-26
Author Wolfgang Konen <wolfgang.konen@th-koeln.de> [aut], Samineh Bagheri <samineh.bagheri@th-koeln.de> [aut,cre], Patrick Koch [aut], Thomas Baeck <t.h.w.baeck@liacs.leidenuniv.nl> [aut]
Maintainer Samineh Bagheri <samineh.bagheri@th-koeln.de>
Description Performs surrogate-assisted optimization for expensive black-box constrained problems.
License GPL (>= 2)
Depends R (>= 2.14.0),
Suggests nloptr, FNN, MASS, dfoptim, DEoptim, lhs, rgl, grDevices, scales, numDeriv, pracma, reshape2, data.table
Imports testit, methods, mgcv, R6
Collate 'cobraInit.R' 'cobraPhaseI.R' 'cobraPhaseII.R' 'debugModel.R' 'defaultDebugRBF.R' 'defaultRI.R' 'defaultSAC.R' 'defaultTR.R' 'defaultCA.R' 'drawSurrogate3d.R' 'evalReal.R' 'fnArchive.R' 'getPredY.R' 'initialHjk.R' 'innerFuncs.R' 'isres2.R' 'modifyEquCons.R' 'modelSelection.R' 'multiRunPlot.R' 'multiRunPlot_2.R' 'multiCOBRA.R' 'nmkb2.R' 'RbfInter.R' 'repairChootinan.R' 'repairInfeasRI2.R' 'SACOBRA.R' 'startCobra.R' 'trainSurrogates.R' 'trustRegion.R' 'updateSaveCobra.R' 'Gproblems.R'

RoxygenNote 7.1.0
NeedsCompilation no
Repository CRAN
Date/Publication 2020-03-26 16:10:02 UTC

R topics documented:

SACOBRA-package .................................................. 2
SACOBRA-package

Description

Self-adjusting Constrained Optimization with RBF Surrogates

Details
SACOBRA is a package for numeric constrained optimization of expensive black-box functions under severely limited budgets. The problem to solve is:

\[
\text{Minimize } f(\vec{x}), \vec{x} \in [\vec{a}, \vec{b}] \subset \mathbb{R}^d \\
\text{subject to } g_i(\vec{x}) \leq 0, i = 1, \ldots, m \\
h_j(\vec{x}) = 0, j = 1, \ldots, r.
\]

SACOBRA is an extension of the COBRA algorithm by Regis (R. Regis: "Constrained optimization by radial basis function interpolation for high-dimensional expensive black-box problems with infeasible initial points", Engineering Optimization, Taylor & Francis, 46, p. 218-243, 2013)

These extensions include:
1) A repair algorithm for infeasible solutions,
2) an algorithm for handling equality constraints,
3) several internal optimizers and several initial design generation methods,
4) self-adjusting random restart algorithm,
5) self-adjusting logarithmic transform for objective functions with large output ranges,
6) range normalization of constraint functions,
7) self-adjusting DRC (distance requirement cycle) selection,
8) online model selection to select the best type of RBF for objective and constraint functions,
9) online whitening for unconstrained optimization of functions with high conditioning.

(Please note that the online whitening implementation is still underway and at this stage it is not recommended to be applied to expensive problems)

SACOBRA performs optimization with a minimum of true function evaluations. It has proven to work well on problems with high dimensions (e.g. d=124) and many constraints (e.g. 60). It is usable for all kind of numerical optimization of continuous functions, but not for combinatorial optimization.

For more details see:


The main entry point functions are `cobraInit` and `startCobra`. See `cobraInit` for an overview of adjustable SACOBRA-parameters. Examples are found in:

- `startCobra`: solve a 13d-problem with 9 inequality constraints (G01)
- `cobraInit`: a problem with equality constraint
- `cobraPhaseII`: unconstrained sphere problem
- `multiCOBRA`: solve G11 problem nrun=4 times
- `COP`: load and solve G24, load and solve the scalable problem G03 with d=3

**Author(s)**

Samineh Bagheri (<Samineh.Bagheri@th-koeln.de>),
Wolfgang Konen (<Wolfgang.Konen@th-koeln.de>),
Patrick Koch, Thomas Baeck (<t.h.w.baeck@liacs.leidenuniv.nl>)

**References**

http://lwibs01.gm.fh-koeln.de/blogs/ciop/research/monrep/

---

**cobraInit**

*Initial phase for SACOBRA optimizer*

**Description**

In this phase the important parameters are set and the initial design population are evaluated on the real function. The problem to solve is:

Minimize \( f(\vec{x}), \vec{x} \in [\vec{a}, \vec{b}] \subset \mathbb{R}^d \)

subject to

\[ g_i(\vec{x}) \leq 0, i = 1, \ldots, m \]

\[ h_j(\vec{x}) = 0, j = 1, \ldots, r. \]
Usage

cobraInit(
    xStart,
    fn,
    fName,
    lower,
    upper,
    feval,
    initDesign = "LHS",
    initDesPoints = 2 * length(xStart) + 1,
    initDesOptP = NULL,
    initBias = 0.005,
    skipPhaseI = TRUE,
    seqOptimizer = "COBYLA",
    seqFeval = 1000,
    seqTol = 1e-06,
    ptail = TRUE,
    squares = TRUE,
    conTol = 0,
    DOSAC = 1,
    sac = defaultSAC(DOSAC),
    repairInfeas = FALSE,
    ri = defaultRI(),
    RBFmodel = "cubic",
    RBFwidth = -1,
    GaussRule = "One",
    widthFactor = 1,
    RBFrho = 0,
    MS = defaultMS(),
    equHandle = defaultEquMu(),
    rescale = TRUE,
    newlower = -1,
    newupper = 1,
    XI = DRCL,
    TrustRegion = FALSE,
    TRlist = defaultTR(),
    conditioningAnalysis = defaultCA(),
    penaF = c(3, 1.7, 3e+05),
    sigmaD = c(3, 2, 100),
    constraintHandling = "DEFAULT",
    verbose = 1,
    verboseIter = 10,
    DEBUG_RBF = defaultDebugRBF(),
    DEBUG_TR = FALSE,
    DEBUG_TRU = FALSE,
    DEBUG_RS = FALSE,
    DEBUG_XI = FALSE,
    trueFuncForSurrogates = FALSE,
saveIntermediate = FALSE,
saveSurrogates = FALSE,
epsilonInit = NULL,
epsilonMax = NULL,
solu = NULL,
cobraSeed = 42
)

Arguments

xStart a vector of dimension d containing the starting point for the optimization problem

fn objective and constraint functions: fn is a function accepting a d-dimensional vector \( \vec{x} \) and returning an \((1+m+r)\)-dimensional vector \( c(f, g_1, \ldots, g_m, h_1, \ldots, h_r) \)

fName the results of cobraPhaseII are saved to <fname>.Rdata

lower lower bound \( \vec{a} \) of search space, same dimension as xStart

upper upper bound \( \vec{b} \) of search space, same dimension as xStart

feval maximum number of function evaluations

initDesign ["LHS"] one out of ["RANDOM","LHS","BIASED","OPTIMIZED","OPTBIASED"]

initDesPoints \([2*d+1]\) number of initial points, must be smaller than feval

initDesOptP \([\text{NULL}]\) only for initDesign=="OPTBIASED": number of points for the "OPT" phase. If NULL, take initDesPoints.

initBias \([0.005]\) bias for normal distribution in "OPTBIASED" and "BIASED"

skipPhaseI \([\text{TRUE}]\) if TRUE, then skip cobraPhaseI

seqOptimizer ["COBYLA"] string defining the optimization method for COBRA phases I and II, one out of ["COBYLA","ISRES","HJKB","NMKB","ISRESCOBY"]

seqFeval \([1000]\) maximum number of function evaluations on the surrogate model

seqTol \([1e-6]\) convergence tolerance for sequential optimizer, see param tol in nmkb or param control$xtol_rel in cobyla

ptail \([\text{TRUE}]\) TRUE: with, FALSE: without polynomial tail in trainRBF

squares \([\text{TRUE}]\) set to TRUE for including the second order polynomials in building the fitness and constraint surrogates in trainRBF

conTol \([0.0]\) constraint violation tolerance

DOSAC \([1]\) set one out of \([0|1|2]\).
0: COBRA-R settings,
1: SACOBRA settings,
2: SACOBRA settings with fewer parameters.
The precise settings are documented in defaultSAC.

sac \([\text{defaultSAC(DOSAC)}]\) list with other parameters for SACOBRA.

repairInfeas \([\text{FALSE}]\) if TRUE, trigger the repair of appropriate infeasible solutions

ri \([\text{defaultRI()}]\) list with other parameters for repairInfeasRI2

RBFmodel ["cubic"] a string for the type of the RBF model, "cubic", "Gaussian" or "MQ"
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBFwidth</td>
<td>[-1]</td>
<td>Only relevant for Gaussian RBF model. Determines the width $\sigma$. For more details see parameter width in trainGaussRBF in RBFInter.R.</td>
</tr>
<tr>
<td>GaussRule</td>
<td>[&quot;One&quot;]</td>
<td>Only relevant for Gaussian RBF model, see trainGaussRBF.</td>
</tr>
<tr>
<td>widthFactor</td>
<td>[1.0]</td>
<td>Only relevant for Gaussian RBF model. Additional constant factor applied to each width $\sigma$.</td>
</tr>
<tr>
<td>RBFrho</td>
<td>[0.0]</td>
<td>Experimental: 0: interpolating, &gt; 0, approximating (spline-like) Gaussian RBFs.</td>
</tr>
<tr>
<td>MS</td>
<td>[defaultMS()]</td>
<td>List of online model selection parameters described in defaultMS. If $MSactive = TRUE$ then the type of RBF models for each function will be selected automatically and the RBFmodel parameter becomes irrelevant.</td>
</tr>
<tr>
<td>equHandle</td>
<td>[defaultEquMu()]</td>
<td>List with of parameters for equality constraint handling described in defaultEquMu(). equHandle$active$ is set to TRUE by default.</td>
</tr>
<tr>
<td>rescale</td>
<td>[TRUE]</td>
<td>If TRUE, transform the input space from $[lower, upper]$ to hypercube $[newlower, newupper]^d$.</td>
</tr>
<tr>
<td>newlower</td>
<td>[-1]</td>
<td>Lower bound of each rescaled input space dimension, if rescale==TRUE.</td>
</tr>
<tr>
<td>newupper</td>
<td>[+1]</td>
<td>Upper bound of each rescaled input space dimension, if rescale==TRUE.</td>
</tr>
<tr>
<td>TrustRegion</td>
<td>[FALSE]</td>
<td>If TRUE, perform trust region algorithm trustRegion.</td>
</tr>
<tr>
<td>TRlist</td>
<td>[defaultTR()]</td>
<td>A list of parameters, needed only in case TrustRegion==TRUE.</td>
</tr>
<tr>
<td>conditioningAnalysis</td>
<td>[defaultCA()]</td>
<td>A list with setting for the objective function conditioning analysis and online whitening.</td>
</tr>
<tr>
<td>penaF</td>
<td>[c(3,1.7,3e5)]</td>
<td>Parameters for dynamic penalty factor (fct subProb in cobraPhaseII): c(start,augment,max), only relevant if seqOptimizer==HJKB or seqOptimizer==NMKB</td>
</tr>
<tr>
<td>sigmaD</td>
<td>[c(3,2.0,100)]</td>
<td>Parameters for dynamic distance factor (fct subProb in cobraPhaseII): c(start,augment,max), only relevant if seqOptimizer==HJKB or seqOptimizer==NMKB</td>
</tr>
<tr>
<td>constraintHandling</td>
<td>[&quot;DEFAULT&quot;]</td>
<td>Other choices: &quot;JOINESHOUCK&quot;, &quot;SMITHTATE&quot;, &quot;COIT&quot;, &quot;BAECKKHURI&quot;; experimental, only relevant if seqOptimizer==HJKB or seqOptimizer==NMKB see the code in function subProb in cobraPhaseII)</td>
</tr>
<tr>
<td>verbose</td>
<td>[1]</td>
<td>Set one out of [0</td>
</tr>
<tr>
<td>verboseIter</td>
<td>[10]</td>
<td>An interger value. Printing the summarized results after each verboseIter iterations.</td>
</tr>
<tr>
<td>DEBUG_RBF</td>
<td>[defaultDebugRBF()]</td>
<td>List with settings for visualization RBF (only for d==2)</td>
</tr>
<tr>
<td>DEBUG_TR</td>
<td>[FALSE]</td>
<td>Prints information about trust region status and visualisation for d==2 (coming soon)</td>
</tr>
<tr>
<td>DEBUG_TRU</td>
<td>[FALSE]</td>
<td>Visualize trust-region RBF (only for dimension==2)</td>
</tr>
<tr>
<td>DEBUG_RS</td>
<td>[FALSE]</td>
<td>Prints the RS probability in each iteration in the console</td>
</tr>
<tr>
<td>DEBUG_XI</td>
<td>[FALSE]</td>
<td>If TRUE, then print in cobraPhaseII extra debug information: xStart in every iteration to console and add some extra debug columns to cobra$df</td>
</tr>
<tr>
<td>trueFuncForSurrogates</td>
<td>[FALSE]</td>
<td>If TRUE, use the true (constraint &amp; fitness) functions instead of surrogates (only for debug analysis)</td>
</tr>
</tbody>
</table>
saveIntermediate
[FALSE] if TRUE, then cobraPhaseII saves intermediate results in dir 'results/' (create it, if necessary)

saveSurrogates
[FALSE] if TRUE, then cobraPhaseII returns the last surrogate models in cobra$fitnessSurrogate and cobra$constraintSurrogates

epsilonInit
[NULL] initial constant added to each constraint to maintain a certain margin to boundary

epsilonMax
[NULL] maximum for constant added to each constraint

solu
[NULL] the best-known solution (only for diagnostics). This is normally a vector of length d. If there are multiple solutions, it is a matrix with d columns (each row is a solution). If NULL, then the current best point will be used in cobraPhaseII. solu is given in original input space.

cobraSeed
[42] seed for random number generator

Details
If epsilonInit or epsilonMax are NULL on input, then cobra$epsilonInit and cobra$epsilonMax, resp., are set to 0.005*l where l is the smallest side of the search box.

Note that the parameters penaF, sigmaD, constraintHandling are only relevant for penalty-based internal optimizers nmkb or HJKB. They are NOT relevant for default optimizer cobyla.

Although the software was originally designed to handle only constrained optimization problems, it can also address unconstrained optimization problems

How to code which constraint is equality constraint? - Function fn should return an (1 + m + r)-dimensional vector with named elements. The first element is the objective, the other elements are the constraints. All equality constraints should carry the name equ. (Yes, it is possible that multiple elements of a vector have the same name.)

Value
cobra, an object of class COBRA, this is a (long) list containing most of the argument settings (see above) and in addition (among others):

A
(feval x dim)-matrix containing the initial design points in input . space. If rescale==TRUE, all points are in rescaled input space.

Fres
a vector of the objective values of the initial design points

Gres
a matrix of the constraint values of the initial design points

nConstraints
the total number m + r of constraints

Tfeas
the threshold parameter for the number of consecutive iterations that yield feasible solutions before margin epsilon is reduced

Tinfeas
the threshold parameter for the number of consecutive iterations that yield infeasible solutions before margin epsilon is increased

numViol
number of constraint violations

maxViol
maximum constraint violation

trueMaxViol
maximum constraint violation
trustregX A vector of all refined solutions generated by trust region algorithm (see trustRegion)

Note that cobra$Fres, cobra$fbest, cobra$fbestArray and similar contain always the objective values of the original function cobra$fn[1]. (The surrogate models may be trained on a plog-transformed version of this function.)

Author(s)
Wolfgang Konen, Samineh Bagheri, Patrick Koch, Cologne University of Applied Sciences

See Also
startCobra, cobraPhaseI, cobraPhaseII

Examples

## Initialize cobra. The problem to solve is the sphere function sum(x^2)
## with the equality constraint that the solution is on a circle with
## radius 2 and center at c(1,0).
##
d=2
##
fName="onCircle"
cobra <- cobraInit(xStart=rep(5,d), fName=fName,
            fn=function(x){c(obj=sum(x^2),equ=(x[1]-1)^2+(x[2]-0)^2-4)},
            lower=rep(-10,d), upper=rep(10,d), feval=40)

## Run sacobra optimizer
cobra <- cobraPhaseII(cobra)

## The true solution is at solu = c(-1,0) (the point on the circle closest
## to the origin) where the true optimum is fn(solu)[1] = optim = 1
## The solution found by SACOBRA:
print(getXbest(cobra))
print(getFbest(cobra))

## Plot the resulting error (best-so-far feasible optimizer result - true optimum)
## on a logarithmic scale:
optim = 1
plot(abs(cobra$df$Best-optim),log="y",type="l",ylab="error",xlab="iteration",main=fName)

---

**cobraPhaseI**  
Find a feasible solution.

**Description**

Find a feasible solution using the COBRA optimizer phase I by searching new infill points. Please note that this phase can be skipped by setting the cobra$skipPhaseI parameter to TRUE in the initialization phase cobraInit()
Usage

cobraPhaseII(cobra)

Arguments

cobra can be an object of class COBRA, this is a (long) list containing all settings from cobraInit

Value

cobra, an object of class COBRA

Author(s)

Wolfgang Konen, Samineh Bagheri, Patrick Koch, Cologne University of Applied Sciences

See Also

cobraPhaseII, cobraInit

---

**cobraPhaseII**

*Improve the feasible solution by searching new infill points*

**Description**

Improve the feasible solution using the SACOBRA optimizer phase II by searching new infill points with the help of RBF surrogate models. May be even called if no feasible solution is found yet, then phase II will try to find feasible solutions.

The problem to solve iteratively is:

\[
\begin{align*}
\text{Minimize} & \quad f(\bar{x}), \bar{x} \in [\bar{a}, \bar{b}] \subset \mathbb{R}^d \\
\text{subject to} & \quad g_i(\bar{x}) \leq 0, i = 1, \ldots, m \\
& \quad h_j(\bar{x}) = 0, j = 1, \ldots, r.
\end{align*}
\]

In this phase the main optimization steps are repeated in a loop as long as the budget is not exhausted. In every iteration the surrogate models are updated and an optimization on the surrogates is done in order to find a better feasible solution.

**Usage**

cobraPhaseII(cobra)

**Arguments**

cobra can be an object of class COBRA, this is a (long) list containing all settings from cobraInit
Value

cobra, an object of class COBRA from cobraInit, enhanced here by the following elements (among others):

fn  function accepting a d-dimensional vector \( \tilde{x} \) and returning an \((1+m+r)\)-vector \( c(f, g_1, \ldots, g_m, h_1, \ldots, h_r) \). This function may be a rescaled and plog-transformed version of the original fn passed into cobraInit. The original fn is stored in cobra$originalFn.

df  data frame with summary of the optimization run (see below)

df2 data frame with additional summary information (see below)

dftr data frame with additional summary information for TR (see below)

A  \((feval \times d)\)-matrix containing all evaluated points in input space. If rescale==TRUE, all points are in rescaled input space.

Fres a vector of the objective values of all evaluated points

Gres a \((feval \times m)\)-matrix of the constraint values of all evaluated points

predC a \((feval \times m)\)-matrix with the prediction of cobra$constraintSurrogates at all evaluated points

fbest the best feasible objective value found

xbest the point in input space yielding the best feasible objective value

ibest the corresponding iteration number (row of cobra$df, of cobra$A)

PLOG If TRUE, then the objective surrogate model is trained on the plog-transformed objective function.

Note that cobra$Fres, cobra$fbest, cobra$fbestArray and similar contain always the objective values of the original function cobra$fn[1]. (The surrogate models may be trained on a plog-transformed version of this function.)

feval = cobra$feval is the maximum number of function evaluations.

The data frame cobra$df contains one row per iteration with columns

iter  iteration index

y  true objective value Fres

predY surrogate objective value. Note: The surrogate may be trained on plog-transformed training data, but predY is transformed back to the original objective range. NA for the initial design points.

predSolu surrogate objective value at best-known solution cobra$solu, if given. If cobra$solu is NULL, take the current point instead. Note: The surrogate may be trained on plog-transformed training data, but predSolu is transformed back to the original objective range. NA for the initial design points.

feasible boolean indicating the feasibility of infill point

feasPred boolean indicating if each infill point is feasible for cobra$constraintSurrogates

nViolations number of violated constraints

maxViolation maximum constraint violation.
**FEval** number of function evaluations in sequential optimizer. NA if it was a repair step

**Best** ever-best feasible objective value \( f_{\text{best}} \). As long as there is no feasible point, take among those with minimum number of violated constraints the one with minimum \( F_{\text{res}} \).

**optimizer** e.g. "COBYLA"

**optimizationTime** in sec

**conv** optimizer convergence code

**dist** distance of the current point (row of `cobra$A`) to the true solution `cobra$solu` in rescaled space. If there is more than one solution, take the one which has the minimum distance element (since this is the solution to which the current run converges).

**distOrig** same as **dist**, but in original space

**XI** the DRC element used in the current iteration

**seed** the used seed in every run

The data frame `cobra$df2` contains one row per phase-II-iteration with columns

**iter** iteration index

**predY** surrogate objective value. Note: The surrogate may be trained on plog-transformed training data, but \( \text{predY} \) is transformed back to the original objective range. NA for the initial design points.

**predVal** surrogate objective value + penalty

**predSolu** surrogate objective value at true solution (see `cobra$df$predSolu`) **predSoluPenal** surrogate objective value + penalty at true solution (only diagnostics)

**sigmaD** the `sigmaD` element used in the current iteration (see `cobraInit`)

**penaF** penalty factor used in the current iteration (see `cobraInit`)

**XI** the DRC element used in the current iteration

**EPS** the current used margin for constraint function modeling (see `epsilonInit` in `cobraInit`)

**Author(s)**
Wolfgang Konen, Samineh Bagheri, Patrick Koch, Cologne University of Applied Sciences

**See Also**
`cobraPhaseI`, `cobraInit`

**Examples**

```r
## Initialize cobra. The problem to solve is the unconstrained sphere function \( \text{sum}(x^2) \).
## In version 1.1 and higher there is no need for defining a dummy constraint function for the unconstrained problems
d=2
fName="sphere"
cobra <- cobraInit(xStart=rep(5,d), fName=fName, fn=function(x){c(obj=sum(x^2))}, lower=rep(-10,d), upper=rep(10,d), feval=40)
```
## Run cobra optimizer

cobra <- cobraPhaseII(cobra)

## The true solution is at solu = c(0,0)
## where the true optimum is fn(solu)[1] = optim = 0
## The solution found by SACOBRA:
print(getXbest(cobra))
print(getFbest(cobra))

## Plot the resulting error (best-so-far feasible optimizer result - true optimum)
## on a logarithmic scale:
optim = 0
plot(cobra$df$Best-optim,log="y",type="l",ylab="error",xlab="iteration",main=fName)

### COP: Constraint Optimization Problem Benchmark (G Function Suite)

**Description**

COP is an object of class `R6ClassGenerator` which can be used to access G problems (aka G functions) implementations in R, by simply generating a new instance of COP for each G function `problem<-COP.new("problem")`. The COP instances have the following useful attributes:

- **name**: name of the problem given by the user
- **dimension**: dimension of the problem. For the scalable problems G02 and G03, the dimension should be given by users, otherwise it will be set automatically.
- **lower**: lower boundary of the problem
- **upper**: upper boundary of the problem
- **fn**: the COP function which can be passed to SACOBRA. (see fn description in `cobraInit`)
- **nConstraints**: number of constraints
- **xStart**: The suggested optimization starting point
- **solv**: the best known solution, (only for diagnostics purposes)
- **info**: information about the problem

G function suite is a set of 24 constrained optimization problems with various properties like dimension, number of equality/inequality constraint, feasibilty ratio, etc. Although these problems were introduced as a suite in a technical report at CEC 2006, many of them have been used by different authors earlier.

Methods

Public methods:

• COP$new()
• COP$clone()

Method new():

Usage:
COP$new(name, dimension)

Method clone(): The objects of this class are cloneable with this method.

Usage:
COP$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

Author(s)

Samineh Bagheri, Wolfgang Konen

Examples

## creating an instance for G24 problem
G24 <- COP$new("G24")

## initializing SACOBRA
cobra <- cobraInit(xStart=G24$lower, fName=G24$name,
                   fn=G24$fn,
                   lower=G24$lower, upper=G24$upper, feval=25)

## Run sacobra optimizer
cobra <- cobraPhaseII(cobra)

## The true solution is at solu = G24$solu
## The solution found by SACOBRA:
print(getXbest(cobra))
print(getFbest(cobra))
plot(abs(cobra$df$Best-G24$fn(G24$solu)[1]),log="y",type="l",
     ylab="error",xlab="iteration",main=G24$name)

## creating an instance for G03 in 2-dimensional space
G03 <- COP$new("G03",2)

## Initializing sacobra
cobra <- cobraInit(xStart=G03$lower, fn=G03$fn,
                   fName=G03$name, lower=G03$lower, upper=G03$upper, feval=40)
Default settings for online whitening functionality

Description

Sets default values for the online whitening functionality in order to handle functions with high conditioning. With the call `setOpts(cobra$CA, defaultCA())` it is possible to extend a partial list `cobra$CA` to a list containing all CA-elements (the missing ones are taken from `defaultCA()`).

As RBF interpolations face severe difficulties to deliver reasonable models for functions with high conditioning, we try to transform the function with high conditioning \( f(\vec{x}) \) to a better conditioned one \( g(\vec{x}) \) which is easier to model.

\[
g(\vec{x}) = f(M(\vec{x} - \vec{x}_c))
\]

A possible transformation matrix \( M \) is the squared inverse of the Hessian matrix \( H^{-0.5} \), assuming that \( M \) is chosen with the following assumption:

\[
\frac{\partial^2 g(\vec{x})}{\partial \vec{x}^2} = I
\]

Usage

`defaultCA()`

Details

The current version is only relevant for unconstrained problems. It this stage it is not recommended to apply the online whitening to expensive optimization problems as it demands large number of function evaluations. Every online whitening call demands \( 4d^2 + 4d \) function evaluations.

Value

`CA`, a list of the following elements:

- **active**: Set to TRUE if an online whitening of the fitness function is desired
- **HessianType**: ["real"] You can choose if the Hessian matrix is evaluated on the real function or on the surrogate model ["real", "surrogate"]: Please note that the determination of Hessian matrix on the real function at each point costs \( 4d^2 + 4d \) real function evaluations.
- **ITERS**: \([\text{seq}(10,500,10)]\), pass a vector of integers to this parameter then the Hessian matrix will be updated only in the given iterations, we recommended applying the online-whitening each 10 iterations after the \( 10*d \) initial iterations. `seq(10*d, maxIter ,10)`, where \( d \) is the dimensionality of the optimization problem. If set as the character "all" then the Hessian matrix will be updated in each iteration and whitening procedure will be repeated.
- **alpha**: [1] you can assign any real value to this parameter. Only values between 0 to 2 are suggested. This value is used in order to modify the transformation center \( tCenter \) as follows: \( \text{xbest}+\text{alpha}*(\text{grad}) \), and \( \text{grad} \) is the direction of the last improvement.
See Also

set0pts

defaultDebugRBF

Default settings for debug visualization RBF (only for d==2)

Description

Sets default values for debug visualization RBF of SACOBRA.

Usage

defaultDebugRBF()

Value

DEBUG_RBF a list of the following elements:

active

If set to TRUE then debugVisualizeRBF is called every DEBUG_RBF$every iterations

overlayTrueZ

If set to TRUE overlay the true objective function

DO_SNAPSHOT

do rgl.snapshot every DEBUG_RBF$every iteration and store it in sprintf("images.d/%s-%03d.png",cobra$fname,npts)

every

Frequency of calling the debugVisualizeRBF function

See Also

defaultEquMu

defaultEquMu

Default settings for equality handling mechanism

Description

Sets suitable defaults for the equality handling part of SACOBRA.

The EH technique transforms each equality constraint \( h(\bar{x}) = 0 \) into two inequality constraints \( h(\bar{x}) - \mu < 0 \) and \( -h(\bar{x}) - \mu < 0 \) with an adaptively decaying margin \( \mu \).

If refine parameter is set to TRUE, then a refine mechanism is applied to shift the best found solution within the equality margin \( \mu \) toward the feasible subspace by minimizing the sum of squared constraint surrogates with a conjugate gradient method.

\[
\text{Minimize } \sum_i (\max(0, g_i(x))^2) + \sum_j (h_j(x))^2
\]
Usage

defaultEquMu()

Details

With the call `setOpts(equHandle, defaultEquMu())` it is possible to extend a partial list `equHandle` list which is set by user to a list containing all `equHandle`-elements (the missing ones are taken from `defaultEquMu()`). These settings are used by `cobraInit` for initializing the equality margin $\mu$ and by the internal functions `updateCobraEqu` and `modifyMu`. The minimization step of refine mechanism is done by L-BFGS-B method in `optim` function from `stats` package.

Value

equHandle, a list with the following elements:

active  
[TRUE] if set to TRUE the equality-handling (EH) technique is activated. The EH technique transforms each equality constraint $h(x) = 0$ into two inequality constraints $h(x) - \mu < 0$ and $-h(x) - \mu < 0$ with an adaptively decaying margin $\mu$.

equEpsFinal  
[1e-07] lower bound for margin $\mu$. `equEpsFinal` should be set to a small but non-zero value (larger than machine accuracy).

initType  
["TAV"] the equality margin $\mu$ can be initialized with one of these approaches:  
["TAV"|"TMV"|"EMV"|"useGrange"]  
TAV: (Total Absolute Violation) takes the median of the sum of violations of the initial population.  
TMV: (Total Maximum Violation) takes the median of the maximum violation of the initial population  
EMV: takes the median of the maximum violation of equality constraints of the initial population  
useGrange: takes the average of the ranges of the equality constraint functions

epsType  
["SAexpFunc"] type of the function used to modify margin $\mu$ during the optimization process can be one of ["SAexpFunc"|"expFunc"|"Zhang"|"CONS"]. see `modifyMu`.

dec  
[1.5] decay factor for margin $\mu$. see `modifyMu`

refine  
[TRUE] enables the refine mechanism f the equality handling mechanism.

refineMaxit  
maximum number of refine iterations used in the refine step. Note that the refine step runs on the surrogate models and does not impose any extra real function evaluation.

See Also

`updateCobraEqu`, `modifyMu`
defaultMS

Default settings for the model-selection part of SACOBRA.

Description

Sets default values for the model-selection part cobra$MS of SACOBRA.
It is shown that different types of RBFs can deliver different qualities in modeling different functions.
Using the online model selection functionality boosted the overall performance of SACOBRA on a
large set of constrained problems. The algorithm trains every function (objective and constraints)
with a given pool of models including different RBF types and width parameters. The type of model
which performs the best in the last iterations \( \text{WinS} \) will be selected for each function. The quality of
the models are determined by different measures of approximation error in each iteration

\[ f(\vec{x}_{\text{new}}) - s(\vec{x}_{\text{new}}) \]

Usage

defaultMS()

Details

With the call \textbf{setOpts}(MS, defaultMS()) it is possible to extend a partial list \textit{MS} to a list containing
all \textit{MS}-elements (the missing ones are taken from \texttt{defaultMS}(\texttt{))).

\textbf{NOTE:} Because of common crash observation, it is not recommended to include Gaussian model
in the set of models especially for problems which require more than 100 function evaluations.

Value

\textit{MS}, a list with the following elements

\begin{itemize}
  \item \textbf{active} [F] If set to TRUE then \texttt{selectModel} calculates the best model for each
       constraint(s)/objective function
  \item \textbf{models} [c("cubic","MQ")]) a set of model types that will be used to build the pool of
       models. Three types of RBF are implemented "cubic", "Gaussian" and "MQ"
       (multiquadric). Users can select one or combination of these models. Users can
       select a set of "width parameters" for "MQ" and "Gaussian" by setting \textbf{widths}
       parameter.
  \item \textbf{widths} [c(0.01,0.1,1,10)] a set of values for width parameter of RBF models. Only
       relevant if \textbf{models} include "Gaussian" or "MQ".
  \item \textbf{freq} [1] controls how often \texttt{selectModel} is called. In every \textbf{freq} iterations all
       the selected models are trained for all constraint/objective function(s)
  \item \textbf{slidingW} [T] when set to FALSE it uses the information taken from all the past iterations
       to assess the quality of the models. When set to TRUE, activates the sliding
       window functionality and it takes the information of the last \textit{WinS} iterations (see
       \textit{WinS}).
\end{itemize}
**defaultRI**

[1] size of the sliding window

**quant**

[3] 3: median, 2:0.25, 4:0.75. The measure used to compare the quality of the model in the last window.

**apply**

[T] if set to FALSE then the selected models are not used during the optimization. Only for debugging purposes.

**considerXI**

[F] If set to T then a subset of the approximation errors which are related to the current (DRC element) are considered to make the model selection decision

**Author(s)**

Samineh Bagheri

**See Also**

`setOpts`

---

**Description**

Sets suitable defaults for the repair-infeasible part of SACOBRA. With the call `setOpts(myRI,defaultRI())` it is possible to extend a partial list `myRI` to a list containing all `ri`-elements (the missing ones are taken from `defaultRI()`)

**Usage**

`defaultRI(repairMargin = 0.01)`

**Arguments**

`repairMargin` [1e-2] repair only solutions whose infeasibility is less than this margin

**Details**

The infeasibility of a solution is its maximum constraint violation (0 for a feasible solution).

**Value**

A list with the following elements:

- **RIMODE**
  [2] one out of {0,1,2,3 } with 0,1: deprecated older versions of RI2, 2: the recommended RI2-case, see `repairInfeasRI2`, 3: Chootinan’s method, see `repairChootinan`

- **eps1**
  [1e-4] include all constraints not eps1-feasible into the repair mechanism

- **eps2**
  [1e-4] selects the solution with the shortest shift among all random realizations which are eps2-feasible
q [3.0] draw coefficients $\alpha_k$ from uniform distribution $U[0, q]$

mmax [1000] draw mmax random realizations

repairMargin repair only solutions whose infeasibility is less than this margin.

repairOnlyFresBetter

[FALSE] if TRUE, then repair only iterates with fitness < so-far-best-fitness + marFres

marFres [0.0] only relevant if repairOnlyFresBetter==TRUE

A solution $x$ is said to be $\epsilon$-feasible for constraint function $f$, if

$$f(x) + \epsilon \leq 0$$

Author(s)

Wolfgang Konen, Cologne University of Applied Sciences

See Also

repairInfeasRI2, repairChootinan

defaultSAC

Default settings for the SACOBRA part of SACOBRA.

Description

Sets suitable defaults for the SACOBRA part of SACOBRA.

With the call setOpts(mySAC, defaultSAC()) it is possible to extend a partial list mySAC to a list containing all sac-elements (the missing ones are taken from defaultSAC()).

Usage

defaultSAC(DOSAC = 1)

Arguments

DOSAC [012] with default 1.
0: COBRA-R settings (turn off SACOBRA),
1: SACOBRA settings,
2: SACOBRA settings with fewer parameters and more online adjustments (aFF and aCF are done parameter free).

Details

For backward compatibility, a logical DOSAC (deprecated) is mapped from FALSE to 0 and from TRUE to 1.
defaultSAC

Value

a list with the following elements (the values in parentheses [ ] are the values for DOSAC=[0|1|2]):

RS flag for random start algorithm [FALSE|TRUE|TRUE]
RStype type of the function to calculate probability to start the internal optimizer with a random starting point[NA|"SIGMOID"|"CONSTANT"] (see function RandomStart in SACOBRA.R)
RSmax maximum probability of a random start when RStype=="SIGMOID" (see RandomStart in SACOBRA.R). If RStype=="CONSTANT" then random start is done with a constant probability determined from mean(c(RSmax,RSmin)) [NA|0.3|0.3]
RSmin minimum probability of a random start when RStype=="SIGMOID" (see RandomStart in SACOBRA.R) [NA|0.05|0.05]
RSAUTO If TRUE then in every iteration where the fraction of feasible points in the population is smaller than 0.05, the RS probability is set to 0.3. [FALSE|FALSE|TRUE]
aDRC flag for automatic DRC adjustment [FALSE|TRUE|TRUE]
aFF flag for automatic objective function transformation [FALSE|TRUE|TRUE]
aCF flag for automatic constraint function transformation [FALSE|TRUE|TRUE]
TFRange threshold, if FRange is larger than TFRange, then apply automatic objective function transformation (see plog). [Inf|1e+05|1]
TGR threshold, if GRatio is larger than TGR, then apply automatic constraint function transformation. GRatio is the ratio "largest GRange / smallest GRange" where GRange is the min-max range of a specific constraint. If TGR < 1, then the transformation is always performed. [Inf|1e+03|1]
Cs If Cs iterations in a row do not improve the ever-best feasible solution, then perform a random restart. [10|10|10]
adaptivePLOG (experimental) flag for objective function transformation with plog, where the parameter pShift is adapted during iterations. [FALSE|FALSE|FALSE]
onlinePLOG flag for online decision making wether use plog or not according to p-effect plog. [FALSE|FALSE|TRUE]
pEffectInit Initial pEffect value when using onlinePLOG. If pEffectInit >= 2 then the initial model is built after plog transformation. [NA|NA|2]

Author(s)

Samineh Bagheri, Cologne University of Applied Sciences

See Also

cobraInit, cobraPhaseII
**defaultTR**

*Default settings for the trust-region part of COBRA.*

**Description**

Sets default values for the trust-region part cobra$TRlist of SACOBRA. With the call `setOpts(myTR,defaultTR())` it is possible to extend a partial list `myTR` to a list containing all `TR`-elements (the missing ones are taken from `defaultTR()`).

**Usage**

`defaultTR()`

**Value**

a list with the following elements

- **shape**
  - "cube"] Shape of the trust region can be chosen between cube or a sphere
    - [cube|sphere]

- **radiMin**
  - [0.01] A value between 0 and 1, minimum fraction of the width of the search space to be used as radius of the trust region

- **radiMax**
  - [0.8] A value between 0 and 1, maximum fraction of the width of the search space to be used as radius of the trust region

- **radiInit**
  - [0.1] Initial radius of trust region

- **center**
  - [cobra$xbest] Center of the trust region can be the current best solution or the new solution[xbest|xnew]

**See Also**

`setOpts`, `trustRegion`

---

**distLine**

*Euclidean distance of x to all xp*

**Description**

Euclidean distance of x to a line of points xp

**Usage**

`distLine(x, xp)`
Arguments

x vector of dimension d
xp n points $x_i$ of dimension d are arranged in (n x d) matrix xp. If xp is a vector, it is interpreted as (n x 1) matrix, i.e. d=1.

Details
distLine is up to 40x faster than using dist and taking only the first row or column of the distance matrix returned.

Value

vector of length n, the Euclidean distances

---

**DRCL**

*Distance Requirement Cycle, long version*

**Description**

Distance Requirement Cycle, long version: c(0.3, 0.05, 0.001, 0.0005, 0.0)

**Usage**

DRCL

**Format**

An object of class numeric of length 5.

---

**DRCS**

*Distance Requirement Cycle, short version*

**Description**

Distance Requirement Cycle, short version: c(0.001, 0.0)

**Usage**

DRCS

**Format**

An object of class numeric of length 2.
Description

Helper for cobraPhaseII: The new iterate xNew, which was found by optimization on the surrogate models, is evaluated on the real function cobra$fn. In the case of equality constraints, evalReal does the additional refine step (see Details).

Usage

evalReal(
  cobra, 
  ev1, 
  xNew, 
  fValue, 
  feval, 
  optimConv, 
  optimTime, 
  currentEps, 
  fitnessSurrogate = cobra$fitnessSurrogate
)

Arguments

cobra an object of class COBRA, this is a (long) list containing all settings from cobraPhaseII
ev1 a list, initially empty, gradually filled by calls to evalReal
xNew the new point, see cobraPhaseII
fValue fitness value estimated for xNew
feval function evaluations on surrogates needed by COBRA optimizer
optimConv see cobraPhaseII
optimTime see cobraPhaseII
currentEps artificial current margin for the equality constraints: A point is said to be artificially feasible, if $h_j(x) - currentEps \leq 0, -h_j(x) - currentEps \leq 0$, for all equality constraints and if it is feasible in the inequality constraints.
fitnessSurrogate [cobra$fitnessSurrogate] see cobraPhaseII

Details

If cobra$sequHandle$active==TRUE, then xNew is first refined: The artificially feasible solution xNew is replaced by a refined solution ev1$xNew. ev1$xNew is created by using optim to minimize the function

$$\sum_i \max(0, g_i(x)) + \sum_j h_j^2(x)$$
Ideally, the refined solution ev1$xNew should be on the equality constraints (within machine accuracy), but there is no guarantee that optim reaches this desired result.

Value

ev1, a list with the following n-dim vectors (n = number of iterations, the last element is from the new iterate / point xNew):

- predY: prediction of fitnessSurrogate at xNew
- predVal: fvalue (fitness + penalty in case of NMKB et al.)
- feval: function evaluations on surrogates needed by COBRA optimizer
- optimizerConvergence: see cobraPhaseII
- optimizationTime: see cobraPhaseII
- predC: prediction of cobra$constraintSurrogates at xNew
- feas: TRUE, if xNew is feasible for the current constraints
- feasPred: TRUE, if xNew is feasible for cobra$constraintSurrogates

In addition, ev1 has these elements:

- xNew: d-dim vector, the new point, refined in the case of equality handling
- xNewEval: cobra$fn(xNew), an (1+nConstraints)-dim vector (objective.constraints)
- newNumViol: scalar, the number of constraint violations (above cobra$conTol) on true constraints from xNewEval
- newNumPred: scalar, the number of constraint violations (above cobra$conTol) on constraint surrogates for xNew
- newMaxViol: scalar, the maximum constraint violation (with currentEps subtracted) on true constraints from xNewEval
- trueMaxViol: scalar, the maximum constraint violation (w/o currentEps subtracted) on true constraints from xNewEval

If cobra$equHandle$active==TRUE, then the last four values are for xNew after the refine step. In this case, the first three elements newNumViol, newNumPred, and newMaxViol refer to the artificially enlarged equality constraints, i.e.

\[ h_j(x) - currentEps \leq 0, \quad -h_j(x) - currentEps \leq 0, \]

and the true inequality constraints \( \max(0, g_i(x)) \). The last element trueMaxViol measures the maximum violation among the true equality constraints \( |h_j(x)| \) and the true inequality constraints \( \max(0, g_i(x)) \).

See Also

cobraPhaseII
forwardRescale  

**Forward Rescaling**

**Description**

Scale vector \( x \) in original space forward to rescaled space (usually \([-1, 1]^d\))

**Usage**

\[ \text{forwardRescale}(x, \text{cobra}) \]

**Arguments**

- \( x \): a vector in the original input space
- \( \text{cobra} \): list from \( \text{cobraInit} \), we need here
  - \( \text{originalL} \): a vector with lower bounds in original input space
  - \( \text{originalU} \): a vector with upper bounds in original input space
  - \( \text{newlower} \): a number, the rescaled lower bound for all dimensions
  - \( \text{newupper} \): a number, the rescaled upper bound for all dimensions

**Value**

\( z \), scaled version of vector \( x \)

**See Also**

- \( \text{inverseRescale} \)

getFbest  

**Return best objective function value**

**Description**

Return the original objective function value at the best feasible solution

**Usage**

\[ \text{getFbest}(\text{cobra}) \]

**Arguments**

- \( \text{cobra} \): an object of class COBRA (see \( \text{cobraInit} \))
GetXbest

Details
Note: We cannot take the best function value via cobra$fn, because this may be modified by plog() or others

Value
the original objective function value at the best feasible solution

See Also
GetXbest

getXbest

Return best feasible solution in original space

Description
Return best feasible solution in original space

Usage
getXbest(cobra)

Arguments
cobra an object of class COBRA (see cobraInit)

Value
the best feasible solution in original space

See Also
getFbest
**intern.archive.env**  *Archiving Environment*

**Description**

intern.archive.env is an independent environment where every evaluated point and its evaluation by the real function are stored in ARCHIVE and ARCHIVEY. This archive stores different values to cobra$A$ and cobra$Fres$ often during debugging and visualisation cases where the real function is evaluated very often for debugging purposes.

**Usage**

intern.archive.env

**Format**

An object of class environment of length 0.

**interpRBF**  *Apply the trained cubic, MQ or Gaussian RBF interpolation to new data for d>1.*

**Description**

Apply the trained cubic, MQ or Gaussian RBF interpolation to new data for d>1.

**Usage**

interpRBF(x, rbf.model)

**Arguments**

- **x**  
  vector holding a point of dimension d

- **rbf.model**  
  trained RBF model (or set of models), see trainCubicRBF or trainGaussRBF

**Value**

value $s(\vec{x})$ of the trained model at $\vec{x}$

- or -

vector $s_j(\vec{x})$ with values for all trained models $j = 1, ..., m$ at $\vec{x}$

**Author(s)**

Wolfgang Konen (<wolfgang.konen@th-koeln.de>)

**See Also**

trainCubicRBF, trainMQRBF, trainGaussRBF, predict.RBFinter
inverseRescale  

**Inverse Rescaling**

**Description**

Scale vector x in rescaled space back to original space

**Usage**

```
inverseRescale(x, cobra)
```

**Arguments**

- `x`: a vector in the rescaled input space (usually \([-1, 1]^d\))
- `cobra`: list from `cobraInit`, we need here
  - `originalL`: a vector with lower bounds in original input space
  - `originalU`: a vector with upper bounds in original input space
  - `newlower`: a number, the rescaled lower bound for all dimensions
  - `newupper`: a number, the rescaled upper bound for all dimensions

**Value**

`z`, inverse rescaling of vector `x`

**See Also**

`forwardRescale`

---

multiCOBRA  

**Perform multiple COBRA runs**

**Description**

Perform multiple COBRA runs. Each run starts with a different seed so that a different start point, a different initial design and different random restarts are choosen.

**Usage**

```
multiCOBRA(
  fn,
  lower,
  upper,
  nrun = 10,
  feval = 200,
  funcName = "GXX",
```

---
fName = paste0("mult-", funcName, ".Rdata"),
path = NULL,
cobra = NULL,
optim = NULL,
target = 0.05,
saveRdata = FALSE,
ylim = c(1e-05, 10000),
plotPDF = FALSE,
startSeed = 41
)

**Arguments**

- **fn**: objective function that is to be minimized, should return a vector of the objective function value and the constraint values
- **lower**: lower bound of search space
- **upper**: upper bound of search space
- **nrun**: [10] number of runs
- **feval**: [200] function evaluations per run
- **funcName**: ["GXX"] name of the problem
- **fName**: the results (dfAll and others) are saved to <fName>.Rdata (only if saveRdata==TRUE)
- **path**: [NULL] optional path
- **cobra**: [NULL] list with COBRA settings. If NULL, initialize cobra with a suitable call to cobraInit.
- **optim**: [NULL] the true optimum (or best known value) of the problem (only for diagnostics)
- **target**: [0.05] a single run meets the target, if the final error is smaller than target
- **saveRdata**: [FALSE] if TRUE, save results (dfAll,optim,target,fName,funcName) on <fName>.Rdata
- **ylim**: the y limits
- **plotPDF**: [FALSE] if TRUE, plot not only to current graphics device but to <fName>.pdf as well
- **startSeed**: [41] after each run the seed is incremented by 1, starting with startSeed

**Details**

Side effect: An error plot showing each run and the mean and median of all runs (see multiRunPlot). The results (dfAll and others) are saved to <fName>.Rdata.

**Value**

- **mres**, a list containing
  - **cobra**: the settings and results from last run
  - **dfAll**: a data frame with a result summary for all runs (see below)
z a vector containing for each run the ever-best feasible objective value
z2 a data frame containing for each run the minimum error (if optim is available)

The data frame dfAll contains one row per iteration with columns (among others)

ffc fitness function calls (i.e. the iterations cobra$iter)
fitVal true fitness function value
fitSur surrogate fitness function value
feas is current iterate feasible on the true constraints?
feval number of evaluations of the internal optimizer on the surrogate functions (NA if it is a repairInfeasible-step)
XI the DRC element used in the current iteration
everBestFeas the ever-best feasible fitness function value
run the number of the current run
X1,X2,... the solution in (original) input space

Author(s)
Wolfgang Konen, Samineh Bagheri, Cologne University of Applied Sciences

See Also
multiRunPlot, cobraPhaseII

Examples

```r
## solve G11 problem nrun times and plot the results of all nrun runs
nrun=4
feval=25

## Defining the constrained problem (G11)
fn <- function(x) {
  y<-x[1]*x[1]+((x[2]-1)^2)
  y<-as.numeric(y)
  g1 <- as.numeric(+x[2] - x[1]^2))
  return(c(objective=y, g1=g1))
}
funcName="G11"
lower<-c(-1,-1)
upper<-c(+1,+1)

## Initializing and running cobra
cobra <- cobraInit(xStart=c(0,0), fn=fn, fName=funcName, lower=lower, upper=upper, 
                 feval=feval, initDesPoints=3*2, DOSAC=1, cobraSeed=1)
mres <- multiCOBRA(fn,lower,upper,nrun=nrun,feval=feval,optim=0.75
```
## There are two true solutions at
## solu1 = c(-sqrt(0.5),0.5) and solu2 = c(+sqrt(0.5),0.5)
## where the true optimum is f(solu1) = f(solu2) = -0.75
## The solution from SACOBRA is close to one of the true solutions:
print(getXbest(mres$cobra))
print(getFbest(mres$cobra))
print(mres$z2)

---

**multiRunPlot**

Plot the results from multiple COBRA runs.

### Description

Plot for each run one black curve 'error vs. iterations' and aggregate the mean curve (red) and the median curve (green) of all runs. 'error' is the distance between the ever-best feasible value and optim.

### Usage

```r
multiRunPlot(
  dfAll,  
  optim = NULL,  
  fName = "multiRun",  
  main = "",  
  xlim = NULL,  
  ylim = c(1e-05, 10000),  
  ylog = TRUE,  
  xlog = FALSE,  
  target = 0.05,  
  plotPDF = FALSE,  
  subPDF = NULL,  
  legendWhere = "topright",  
  absErr = FALSE
)
```

### Arguments

- **dfAll**: the data frame of all runs, obtained with `multiCOBRA` or loaded from .Rdata file
- **optim**: [NULL] the true optimum (or best known value) of the problem (only for diagnostics). If `optim==NULL`, we plot instead of errors the ever-best feasible values.
- **fName**: ["multiRun"] the name of the .Rdata file, printed as subtitle
- **main**: ["""] the name of the problem (e.g. "G01 problem"), printed as title
multiRunPlot_2

- xlim: the x limits
- ylim: the y limits
- ylog: [TRUE] logarithmic y-axis
- xlog: [FALSE] logarithmic x-axis
- target: [0.05] a single run meets the target, if the final error is smaller than target
- plotPDF: [FALSE] if TRUE, plot to 'fName'.pdf
- subPDF: [NULL] optional subdirectory where .pdf should go
- legendWhere: ["topright"]
- absErr: [FALSE] if TRUE, plot abs(error) instead of error.

Details
Print some diagnostic information: final median & mean error, percentage of runs which meet the target (only if optim is available).

Value
z3, a vector containing for each run the ever-best feasible objective value

Author(s)
Wolfgang Konen, Samineh Bagheri, Cologne University of Applied Sciences

See Also
multiRunPlot_2, multiCOBRA, cobraPhaseII

multiRunPlot_2
Plot the results from multiple COBRA runs.

Description
Plot for each run one black curve 'error vs. iterations' and aggregate the mean curve (red) and the median curve (green) of all runs. DIFFERENCE to multiRunPlot: 'error' is the distance of the ever-best feasible point in input space to the true solution solu.

Usage
multiRunPlot_2(
  dfAll,
  solu,
  fName = "multiRun",
  main = "",
  xlim = NULL,
)
multiRunPlot_2

```r
ylim = c(1e-05, 10000),
ylog = TRUE,
xlog = FALSE,
target = 0.05,
plotPDF = FALSE,
subPDF = NULL,
legendWhere = "topright",
absErr = FALSE 
```

**Arguments**

- `dfAll` the data frame of all runs, obtained with `multiCOBRA` or loaded from .Rdata file
- `solu` the true solution in input space of the problem (only for diagnostics).
- `fName` "multiRun" the name of the .Rdata file, printed as subtitle
- `main` "" the name of the problem (e.g. "G01 problem"), printed as title
- `xlim` the x limits
- `ylim` the y limits
- `ylog` [TRUE] logarithmic y-axis
- `xlog` [FALSE] logarithmic x-axis
- `target` [0.05] a single run meets the target, if the final error is smaller than `target`
- `plotPDF` [FALSE] if TRUE, plot to `<fName>.pdf`
- `subPDF` [NULL] optional subdirectory where .pdf should go
- `legendWhere` ["topright"]
- `absErr` [FALSE] if TRUE, plot abs(error) instead of error.

**Details**

Print some diagnostic information: final median & mean error, percentage of runs which meet the target (only if `optim` is available).

**Value**

- `z3`, a vector containing for each run the ever-best feasible objective value

**Author(s)**

Wolfgang Konen, Samineh Bagheri, Cologne University of Applied Sciences

**See Also**

`multiRunPlot, multiCOBRA, cobraPhaseII`
**plog**  
*Monotonic transform*

**Description**

The function is introduced in [Regis 2014] and extended here by a parameter $p_{shift}$. It is used to squash functions with a large range into a smaller range.

Let $y' = (y - p_{shift})$:

$$plog(y) = \ln(1 + y'), \quad \text{if} \quad y' \geq 0$$

$$plog(y) = -\ln(1 - y'), \quad \text{if} \quad y' < 0$$

**Usage**

```r
plog(y, pShift = 0)
```

**Arguments**

- `y`: function argument
- `pShift`: shift

**Value**

$plog(y)$

**See Also**

`plogReverse`

---

**plogReverse**  
*Inverse of plog*

**Description**

Inverse of $plog$

**Usage**

```r
plogReverse(y, pShift = 0)
```

**Arguments**

- `y`: function argument
- `pShift`: shift
predict.RBFinter

Value

\[ plog^{-1}(y) \]

See Also

plog

---

**predict.RBFinter**

*Apply cubic or Gaussian or MQ RBF interpolation*

### Description

Apply cubic or Gaussian or MQ RBF interpolation to a set of new data points for \( d > 1 \).

### Usage

```r
## S3 method for class 'RBFinter'
predict(rbf.model, newdata, ...)
```

### Arguments

- `rbf.model`  
  trained RBF model (or set of models), see `trainCubicRBF` or `trainGaussRBF`

- `newdata`  
  matrix or data frame with \( d \) columns. Each row contains a data point \( x_i, \ i = 1, \ldots, n \)

- `...`  
  (not used)

### Value

- vector of model responses \( s(x_i) \), one element for each data point \( x_i \)
- or -
- if `rbf.model` is a set of \( m \) models, a \( (n \times m) \)-matrix containing in each row the response \( s_j(x_i) \) of all models \( j = 1, \ldots, m \) to \( x_i \)

### Author(s)

Wolfgang Konen (<wolfgang.konen@th-koeln.de>)

### See Also

`trainCubicRBF`, `trainGaussRBF`, `interpRBF`
**repairChootinan**  
*Repair an infeasible solution with the method of Chootinan.*

**Description**


**Usage**

```r
repairChootinan(x, gReal, rbf.model, cobra, checkIt = FALSE)
```

**Arguments**

- `x` an infeasible solution vector $\vec{x}$ of dimension $d$
- `gReal` a vector $(g_1(\vec{x}), \ldots, g_m(\vec{x}), h_1(\vec{x}), \ldots, h_r(\vec{x}))$ holding the real constraint values at $\vec{x}$
- `rbf.model` the constraint surrogate models
- `cobra` parameter list, we need here
  - `lower` lower bounds of search region
  - `upper` upper bounds of search region
  - `ri` a list with all parameters for `repairChootinan`
  - `trueFuncForSurrogate` if TRUE (only for diagnostics), use the true constraint functions instead of the constraint surrogate models `rbf.model`
  - `fn` true functions, only needed in case of `trueFuncForSurrogate==TRUE`
- `checkIt` [FALSE] if TRUE, perform a check whether the returned solution is really feasible. Needs access to the true constraint function `conFunc`

**Value**

$z$, a vector of dimension $d$ with a repaired (hopefully feasible) solution

**Author(s)**

Wolfgang Konen, Cologne University of Applied Sciences

**See Also**

`repairInfeasRI2`, `cobraPhaseII`
Repair an infeasible solution with the method RI2

Description

If the solution $\vec{x}$ is infeasible, i.e. if there is any $i$ or any $j$ such that

$$g_i(\vec{x}) > 0 \text{ or } |h_j(\vec{x})| - \text{currentEps} > 0$$

:

1. Estimate the gradient of the constraint surrogate function(s) (go a tiny step in each dimension in the direction of constraint increase).
2. Take `cobra$ri$mmax` random realizations in the 'feasible parallelepiped' and select among them the best feasible solution, based on the surrogates,
3. Check whether the new solution is for every dimension in the bounds `[cobra$lower, cobra$upper]` of the search region. If not, set the gradient to 0 in these dimensions and re-iterate from step 2.

There is no guarantee but a good chance, that the returned solution $z$ will be feasible.

Usage

repairInfeasRI2(x, gReal, rbf.model, cobra, checkIt = FALSE)

Arguments

- **x**: an infeasible solution vector $\vec{x}$ of dimension $d$
- **gReal**: a vector $(g_1(\vec{x}), \ldots, g_m(\vec{x}), h_1(\vec{x}), \ldots, h_r(\vec{x}))$ holding the real constraint values at $\vec{x}$
- **rbf.model**: the constraint surrogate models
- **cobra**: parameter list, we need here
  - **lower**: lower bounds of search region
  - **upper**: upper bounds of search region
- **ri**: a list with all parameters for repairInfeasRI2, see `defaultRI`
  - **trueFuncForSurrogate**: if TRUE (only for diagnostics), use the true constraint functions instead of the constraint surrogate models `rbf.model`
  - **fn**: true functions, only needed in case of `trueFuncForSurrogate==TRUE`
- **checkIt**: [FALSE] if TRUE, perform a check whether the returned solution is really feasible. Needs access to the true constraint functions.

Details

Value

\( z \), a vector of dimension \( d \) with a repaired (hopefully feasible) solution

Author(s)

Wolfgang Konen, Cologne University of Applied Sciences

See Also

repairChootinan, cobraPhaseII

---

\textbf{rescaleWrapper} \hspace{1cm} \textit{Return a rescaled function}

Description

Return a rescaled function

Usage

\texttt{rescaleWrapper(fn, lower, upper, dimension, newlower, newupper)}

Arguments

\begin{itemize}
  \item \texttt{fn} \hspace{1cm} function with argument \( x \) to be rescaled
  \item \texttt{lower} \hspace{1cm} a vector with lower bounds in original input space
  \item \texttt{upper} \hspace{1cm} a vector with lower bounds in original input space
  \item \texttt{dimension} \hspace{1cm} length of vector lower and upper
  \item \texttt{newlower} \hspace{1cm} a number, the rescaled lower bound for all dimensions
  \item \texttt{newupper} \hspace{1cm} a number, the rescaled upper bound for all dimensions
\end{itemize}

Value

\texttt{newfn}, rescaled version of function \texttt{fn}

See Also

forwardRescale, inverseRescale
setOpts

Merge the options from a partial list and the default list

Description

Merge the options from a partial list and the default list

Usage

setOpts(opts, defaultOpt)

Arguments

- opts: a partial list of options
- defaultOpt: a list with default values for every element

Value

a list combined from opts and defaultOpt where every available element in opts overrides the default. For the rest of the elements the value from defaultOpt is taken. A warning is issued for every element appearing in opts but not in defaultOpt

Author(s)

Samineh Bagheri, Wolfgang Konen, Cologne University of Applied Sciences

See Also

defaultRI, defaultSAC, defaultTR, defaultEquMu

startCobra

Start COBRA (constraint-based optimization) phase I and/or phase II

Description

Start COBRA (constraint-based optimization) phase I and/or phase II for object cobra

Usage

startCobra(cobra)

Arguments

- cobra: initialized COBRA object, i.e. the return value from cobraInit
Value

cobra, an object of class COBRA

See Also

cobraInit, cobraPhaseI, cobraPhaseII

Examples

## solve G01 problem

## defining the constraint problem: G01
fn <- function(x){
  obj <- sum(5*x[1:4]) - (5*sum(x[1:4]*x[1:4])) - (sum(x[5:13]))
  g4 <- -8*x[1] + x[10]
  g5 <- -8*x[2] + x[11]
  g6 <- -8*x[3] + x[12]
  res <- c(obj, g1, g2, g3, g4, g5, g6, g7, g8, g9)
  return(res)
}

fName = "G01"
d = 13
lower = rep(0, d)
upper = c(rep(1, 9), rep(100, 3), 1)
set.seed(1)
xStart <- runif(d, min=lower, max=upper)

## Initializing cobra
cobra <- cobraInit(xStart=xStart, fn=fn, fName=fName, lower=lower, upper=upper,
  feval=55, seqFeval=400, initDesPoints=3*d, DOSAC=1, cobraSeed=1)

cobra <- startCobra(cobra)

## The true solution is at solu = c(rep(1,9),rep(3,3),1)
## where the optimum is f(solu) = optim = -15
## The solutions from SACOBRA is close to this:
print(getXbest(cobra))
print(getFbest(cobra))

## Plot the resulting error (best-so-far feasible optimizer result - true optimum)
## on a logarithmic scale:
trainCubicRBF

Fit cubic RBF interpolation to training data X for d>1.

Description

The model at a point \( z = (z_1, ..., z_d) \) is fitted using n sample points \( x_1, ..., x_n \)

\[
s(z) = \lambda_1 \Phi(||z - x_1||) + ... + \lambda_n \Phi(||z - x_n||) + c_0 + c_1 * z_1 + ... + c_d * z_d
\]

where \( \Phi(r) = r^3 \) denotes the cubic radial basis function. The coefficients \( \lambda_1, ..., \lambda_n, c_0, c_1, ..., c_d \) are determined by this training procedure.

This is for the default case squares==FALSE. In case squares==TRUE there are d additional pure square terms and the model is

\[
s_{sq}(z) = s(z) + c_{d+1} * z_1^2 + ... + c_{d+d} * z_d^2
\]

In case ptail==FALSE the polynomial tail (all coefficients \( c_i \)) is omitted completely.

Usage

```r
trainCubicRBF(
  xp,
  U,
  ptail = TRUE,
  squares = FALSE,
  rho = 0,
  DEBUG2 = FALSE,
  width = NA
)
```

Arguments

- `xp`: n points \( x_i \) of dimension d are arranged in (n x d) matrix xp
- `U`: vector of length n, containing samples \( f(x_i) \) of the scalar function \( f \) to be fitted
  - or -
  (n x m) matrix, where each column 1,...,m contains one vector of samples \( f_j(x_i) \) for the m'th model, j=1,...,m
- `ptail`: [TRUE] flag, see description
- `squares`: [FALSE] flag, see description
- `rho`: [0.0] experimental: 0: interpolating, >0, approximating (spline-like) Gaussian RBFs
- `DEBUG2`: [FALSE] if TRUE, save M and rhs on return value
- `width`: [NA] non relevant for the parameter-free cubic RBF
trainGaussRBF

Details

The linear equation system is solved via SVD inversion. Near-zero elements in the diagonal matrix \( D \) are set to zero in \( D^{-1} \). This is numerically stable for rank-deficient systems.

Value

rbf.model, an object of class RBFinter, which is basically a list with elements:

- **coef** (n+d+1 x m) matrix holding in column m the coefficients for the m'th model: \( \lambda_1, \ldots, \lambda_n, c_0, c_1, \ldots, c_d \). In case squares==TRUE it is an (n+2d+1 x m) matrix holding additionally the coefficients \( c_{d+1}, \ldots, c_{d+d} \).
- **xp** matrix xp
- **d** size of the polynomial tail. If length(d)==0 it means no polynomial tail will be used for the model. In case of ptail==T && squares==F d will be dimension+1 and in case of ptail==T && squares==T d will be 2*dimension+1
- **npts** number n of points \( x_i \)
- **ptail** TRUE or FALSE (see description)
- **squares** TRUE or FALSE (see description)
- **type** "CUBIC"
- **width** NA, irrelevant for the parameter-free cubic RBF

Author(s)

Wolfgang Konen (<wolfgang.konen@th-koeln.de>), Samineh Bagheri (<samineh.bagheri@th-koeln.de>)

See Also

trainGaussRBF, trainMQRBF predict.RBFinter, interpRBF

trainGaussRBF \hspace{1cm} \textit{Fit Gaussian RBF model to training data for d>1.}

Description

The model for a point \( z = (z_1, \ldots, z_d) \) is fitted using n sample points \( x_1, \ldots, x_n \)

\[
s(z) = \lambda_1 \cdot \Phi(||z - x_1||) + \ldots + \lambda_n \cdot \Phi(||z - x_n||) + c_0 + c_1 \cdot z_1 + \ldots + c_d \cdot z_d
\]

where \( \Phi(r) = \exp(-r^2/(2 \cdot \sigma^2)) \) denotes the Gaussian radial basis function with width \( \sigma \). The coefficients \( \lambda_1, \ldots, \lambda_n, c_0, c_1, \ldots, c_d \) are determined by this training procedure. This is for the default case squares==FALSE. In case squares==TRUE there are d additional pure square terms and the model is

\[
s_{sq}(z) = s(z) + c_{d+1} \cdot z_1^2 + \ldots + c_{d+d} \cdot z_d^2
\]

In case ptail==FALSE the polynomial tail (all coefficients \( c_i \)) is omitted completely.

The linear equation system is solved via SVD inversion. Near-zero elements in the diagonal matrix \( D \) are set to zero in \( D^{-1} \). This makes rank-deficient systems numerically stable.
Usage

\texttt{trainGaussRBF(xp, U, ptail = TRUE, squares = FALSE, width, RULE = "One", widthFactor = 1, rho = 0, DEBUG2 = F)}

Arguments

\textbf{xp} \hspace{1cm} n points \(x_i\) of dimension \(d\) are arranged in (n x d) matrix \(xp\)

\textbf{U} \hspace{1cm} vector of length n, containing samples \(u(x_i)\) of the scalar function \(u\) to be fitted
- or -
(n x m) matrix, where each column \(1,...,m\) contains one vector of samples \(u_j(x_i)\) for the \(m\)'th model, \(j=1,...,m\)

\textbf{ptail} \hspace{1cm} [TRUE] flag, see description

\textbf{squares} \hspace{1cm} [FALSE] flag, see 'Description'

\textbf{width} \hspace{1cm} [-1] either a positive real value which is the constant width \(\sigma\) for all Gaussians in all iterations, or -1. If -1, the appropriate width \(\sigma\) is calculated anew in each iteration with one of the rules \texttt{RULE}, based on the distribution of data points \(xp\).

\textbf{RULE} \hspace{1cm} ["One"] one out of ["One" | "Two" | "Three"], different rules for automatic estimation of width \(\sigma\). Only relevant if \texttt{width} = -1,

\textbf{widthFactor} \hspace{1cm} [1.0] additional constant factor applied to each width \(\sigma\)

\textbf{rho} \hspace{1cm} [0.0] experimental: 0.0: interpolating, >0.0, approximating (spline-like) Gaussian RBFs

\textbf{DEBUG2} \hspace{1cm} [FALSE] if TRUE, save \(M\) and \(\text{rhs}\) on return value

Value

\texttt{rbf.model}, an object of class \texttt{RBFinter}, which is basically a list with elements:

\textbf{coef} \hspace{1cm} (n+d+1 x m) matrix holding in column \(m\) the coefficients for the \(m\)'th model: \(\lambda_1,...,\lambda_n, c_0, c_1, ..., c_d\). In case \texttt{squares==TRUE} it is an (n+2d+1 x m) matrix holding additionally the coefficients \(c_{d+1}, ..., c_{d+d}\).

\textbf{xp} \hspace{1cm} matrix \(xp\)

\textbf{d} \hspace{1cm} size of the polynomial tail. If \texttt{length(d)==0} it means no polynomial tail will be used for the model. In case of \(ptail==T \&\& \texttt{squares==F}\) \(d\) will be dimension+1 and in case of \(ptail==T \&\& \texttt{squares==T}\) \(d\) will be 2*dimension+1

\textbf{npts} \hspace{1cm} number n of points \(x_i\)

\textbf{ptail} \hspace{1cm} TRUE or FALSE (see description)
trainMQRBF

squares TRUE or FALSE (see description)
width the calculated width $\sigma$
type "GAUSS"

Author(s)
Wolfgang Konen, Samineh Bagheri

See Also
trainCubicRBF, predict.RBFinter, interpRBF

trainMQRBF
Fit multiquadric RBF model to training data for $d>1$.

Description
The model for a point $z = (z_1, ..., z_d)$ is fitted using $n$ sample points $x_1, ..., x_n$

$$s(z) = \lambda_1 \Phi(||z - x_1||) + ... + \lambda_n \Phi(||z - x_n||) + c_0 + c_1 z_1 + ... + c_d z_d$$

where $\Phi(r) = \sqrt{1 + (r/\sigma)^2}$ denotes the multiquadrics radial basis function with width $\sigma$. The coefficients $\lambda_1, ..., \lambda_n, c_0, c_1, ..., c_d$ are determined by this training procedure.

This is for the default case squares==FALSE. In case squares==TRUE there are $d$ additional pure square terms and the model is

$$s_{sq}(z) = s(z) + c_{d+1} z_1^2 + ... + c_{d+d} z_d^2$$

In case ptail==FALSE the polynomial tail (all coefficients $c_i$) is omitted completely.

Usage

```r
trainMQRBF(
  xp,
  U,
  ptail = TRUE,
  squares = FALSE,
  width,
  RULE = "One",
  widthFactor = 1,
  rho = 0,
  DEBUG2 = F
)
```
Arguments

- **xp**: n points $x_i$ of dimension d are arranged in (n x d) matrix xp
- **U**: vector of length n, containing samples $u(x_i)$ of the scalar function $u$ to be fitted
- **ptail**: [TRUE] flag, see description
- **squares**: [FALSE] flag, see 'Description'
- **width**: [-1] either a positive real value which is the constant width $\sigma$ for all Gaussians in all iterations, or -1. If -1, the appropriate width $\sigma$ is calculated anew in each iteration with one of the rules RULE, based on the distribution of data points xp.
- **RULE**: ["One"] one out of ["One" | "Two" | "Three"], different rules for automatic estimation of width $\sigma$. Only relevant if width = -1,
- **widthFactor**: [1.0] additional constant factor applied to each width $\sigma$
- **rho**: [0.0] experimental: 0.0: interpolating, >0.0, approximating (spline-like) Gaussian RBFs
- **DEBUG2**: [FALSE] if TRUE, save $M$ and rhs on return value

Details

The linear equation system is solved via SVD inversion. Near-zero elements in the diagonal matrix $D$ are set to zero in $D^{-1}$. This makes rank-deficient systems numerically stable.

Value

- **rbf.model**, an object of class `RBFinter`, which is basically a list with elements:
  - **coef**: (n+d+1 x m) matrix holding in column m the coefficients for the m’th model: $\lambda_1, ..., \lambda_n, c_0, c_1, ..., c_d$. In case squares==TRUE it is an (n+2d+1 x m) matrix holding additionally the coefficients $c_{d+1}, ..., c_{d+d}$.
  - **xp**: matrix xp
  - **d**: size of the polynomial tail. If length(d)==0 it means no polynomial tail will be used for the model. In case of ptail==T && squares==F d will be dimension+1 and in case of ptail==T && squares==T d will be 2*dimension+1
  - **npts**: number n of points $x_i$
  - **ptail**: TRUE or FALSE (see description)
  - **squares**: TRUE or FALSE (see description)
  - **width**: the calculated width $\sigma$
  - **type**: "MQ"

Author(s)

Wolfgang Konen, Samineh Bagheri
trustRegion

See Also

trainCubicRBF, predict.RBFinter, interpRBF

trustRegion

Performs trust region refinement

Description

If cobra$TrustRegion==TRUE (see cobraInit), then the trustRegion functionality is applied every iteration in order to refine the best solution so far. This function builds a local model around the best solution and runs a local search in the trust region to refine the best solution and find a better solution in the neighborhood.

Usage

trustRegion(cobra, center = cobra$xbest)

Arguments

cobra an object of class cobra, which is basically a list (see cobraInit)
center [cobra$xbest] the center of the trust region

Value

the modified cobra with new/updated elements

TRDONE logical, is TRUE if there are more than d+1 points in the trusted region and thus surrogates can be trained. Otherwise FALSE.

trustregX if TRDONE==TRUE the refined solution from the trust-region call, otherwise NA

If TRDONE==TRUE the relevant lists and counters (A,Fres,df,...) of cobra will be updated in cobraPhaseII as well.

Author(s)

Samineh Bagheri (<samineh.bagheri@th-koeln.de>)
Index

+ Topic RBF
  SACOBRA-package, 2
+ Topic black-box
  SACOBRA-package, 2
+ Topic constraints
  SACOBRA-package, 2
+ Topic datasets
  COP, 13
  DRCL, 23
  DRCS, 23
  intern.archive.env, 28
+ Topic optimization
  SACOBRA-package, 2
+ Topic package
  SACOBRA-package, 2
+ Topic surrogate
  SACOBRA-package, 2

cobraInit, 4, 4, 9–13, 17, 21, 26, 27, 29, 30, 40, 41, 47
cobraPhaseI, 6, 9, 9, 12, 41
cobraPhaseII, 4, 6–10, 10, 21, 24, 25, 31, 33, 34, 37, 39, 41, 47
cobyla, 6, 8
COP, 4, 13
debugVisualizeRBF, 16
defaultCA, 7, 15
defaultDebugRBF, 16
defaultEquMu, 7, 16, 40
defaultMS, 7, 18
defaultRI, 6, 19, 38, 40
defaultSAC, 6, 20, 40
defaultTR, 7, 22, 40
dist, 23
distLine, 22
DRCL, 23
DRCS, 23
evalReal, 24

forwardRescale, 26, 29, 39
getFbest, 26, 27
getXbest, 27, 27
intern.archive.env, 28
interpRBF, 28, 36, 43, 45, 47
inverseRescale, 26, 29, 39
modifyMu, 17
multiCOBRA, 4, 29, 32–34
multiRunPlot, 30, 31, 32, 33, 34
multiRunPlot_2, 33, 33
nmkb, 6, 8
plog, 9, 11, 21, 35, 35, 36
plogReverse, 35, 35
predict.RBFinter, 28, 36, 43, 45, 47
RandomStart, 21
repairChootinan, 19, 20, 37, 39
repairInfeasRI2, 6, 19, 20, 37, 38
rescaleWrapper, 39
SACOBRA (SACOBRA-package), 2
SACOBRA-package, 2
setOpts, 15–20, 22, 40
startCobra, 4, 9, 40
trainCubicRBF, 28, 36, 42, 45, 47
trainGaussRBF, 7, 28, 36, 43, 43
trainMQRBF, 28, 43, 45
trustRegion, 7, 22, 47
updateCobraEqu, 17