Package ‘CEGO’

December 8, 2019

Version 2.4.0
Title Combinatorial Efficient Global Optimization
Author Martin Zaefferer <mzaefferer@gmail.com>
Maintainer Martin Zaefferer <mzaefferer@gmail.com>
Description Model building, surrogate model
based optimization and Efficient Global Optimization in combinatorial
or mixed search spaces.
License GPL (>= 3)
Type Package
LazyLoad yes
Depends R (>= 3.0.0)
Imports MASS, stats, DEoptim, graphics, quadprog, Matrix, methods,
ParamHelpers, fastmatch
Suggests nloptr
ByteCompile TRUE
Date 2019-12-07
RoxygenNote 7.0.2
NeedsCompilation yes
Repository CRAN
Date/Publication 2019-12-08 16:40:02 UTC

R topics documented:

CEGO-package .......................................................... 3
benchmarkGeneratorFSP ........................................ 4
benchmarkGeneratorMaxCut ..................................... 5
benchmarkGeneratorNKL ......................................... 6
benchmarkGeneratorQAP ......................................... 7
benchmarkGeneratorTSP ......................................... 8
benchmarkGeneratorWT ......................................... 9
correctionCNSD ....................................................... 10
correctionDefinite .............................................. 11
correctionDistanceMatrix ..................................... 12
correctionKernelMatrix ......................................... 13
createSimulatedTestFunction ................................ 14
distanceMatrix .................................................. 16
distanceNumericHamming ...................................... 17
distanceNumericLCstr ......................................... 18
distanceNumericLevenshtein .................................. 18
distancePermutationAdjacency ................................ 19
distancePermutationChebyshev ................................ 20
distancePermutationCos ........................................ 21
distancePermutationEuclidean ................................ 22
distancePermutationHamming .................................. 23
distancePermutationInsert ..................................... 23
distancePermutationInterchange ............................... 24
distancePermutationLCstr ...................................... 25
distancePermutationLee ......................................... 26
distancePermutationLevenshtein .............................. 27
distancePermutationLex ......................................... 28
distancePermutationManhattan ................................ 29
distancePermutationPosition ................................... 30
distancePermutationPosition2 ................................ 31
distancePermutationR ............................................ 32
distancePermutationSwap ....................................... 33
distanceRealEuclidean ......................................... 34
distanceStringHamming ......................................... 34
distanceStringLCstr ............................................. 35
distanceStringLevenshtein ..................................... 36
distanceVector .................................................... 36
infillExpectedImprovement ..................................... 37
is.CNSD ............................................................. 37
is.NSD .............................................................. 39
is.PSD .............................................................. 40
kernelMatrix ....................................................... 41
landscapeGeneratorGaussian .................................. 41
landscapeGeneratorMUL ......................................... 43
landscapeGeneratorUNI .......................................... 44
lexicographicPermutationOrderNumber ....................... 45
modelKriging ...................................................... 46
modelLinear ....................................................... 49
modelRBFN ........................................................ 50
mutationBinaryBitFlip ......................................... 52
mutationBinaryBlockInversion ................................. 53
mutationBinaryCycle ............................................ 53
mutationBinarySingleBitFlip ................................. 54
mutationPermutationInsert .................................... 54
mutationPermutationInterchange .............................. 55
CEGO-package

Description

Combinatorial Efficient Global Optimization
Details

Model building, surrogate model based optimization and Efficient Global Optimization in combinatorial or mixed search spaces. This includes methods for distance calculation, modeling and handling of indefinite kernels/distances.

Package: CEGO
Type: Package
Version: 2.4.0
Date: 2019-12-07
License: GPL (>= 3)
LazyLoad: yes

Acknowledgments

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

Author(s)

Martin Zaefferer <mzaeferrer@gmail.com>

References


See Also

Interface of main function: optimCEGO

benchmarkGeneratorFSP Create Flow shop Scheduling Problem (FSP) Benchmark

Description

Creates a benchmark function for the Flow shop Scheduling Problem.
benchmarkGeneratorMaxCut

Usage

benchmarkGeneratorFSP(a, n, m)

Arguments

a matrix of processing times for each step and each machine
n number of jobs
m number of machines

Value

the function of type \( \text{cost} = f(\text{permutation}) \)

See Also

benchmarkGeneratorQAP, benchmarkGeneratorTSP, benchmarkGeneratorWT

Examples

n=10
m=4
# create a matrix of processing times
A <- matrix(sample(100, replace=TRUE), n, m)
# create FSP objective function
fun <- benchmarkGeneratorFSP(A, n, m)
# evaluate
fun(1:n)
fun(n:1)

benchmarkGeneratorMaxCut

MaxCut Benchmark Creation

Description

Generates MaxCut problems, with binary decision variables. The MaxCut Problems are transformed to minimization problems by negation.

Usage

benchmarkGeneratorMaxCut(N, A)

Arguments

N length of the bit strings
A The adjacency matrix of the graph. Will be created at random if not provided.
benchmarkGeneratorNKL

Value

the function of type cost=f(bitstring). Returned fitness values will be negative, for purpose of minimization.

Examples

fun <- benchmarkGeneratorMaxCut(N=6)
fun(c(1,0,1,1,0,0))
fun(c(1,0,1,0,1))
fun(c(0,1,0,0,1,1))

fun <- benchmarkGeneratorMaxCut(A=matrix(c(0,1,0,1,1,0,1,0,1,0,1,0,1,0,1,0),4,4))
fun(c(1,0,1,0))
fun(c(1,0,1,1))
fun(c(0,1,0,1))

________________________________________________________________________

benchmarkGeneratorNKL  NK-Landscape Benchmark Creation

Description

Function that generates a NK-Landscapes.

Usage

benchmarkGeneratorNKL(N = 10, K = 1, PI = 1:K, g)

Arguments

N       length of the bit strings
K       number of neighbours contributing to fitness of one position
PI      vector, giving relative positions of each neighbour in the bit-string
g      set of fitness functions for each possible combination of string components. Will be randomly determined if not specified. Should have N rows, and 2^(K+1) columns.

Value

the function of type cost=f(bitstring). Returned fitness values will be negative, for purpose of minimization.
**Examples**

```r
fun <- benchmarkGeneratorQAP(6,2)
fun(c(1,0,1,1,0,0))
fun(c(1,0,1,1,0,1))
fun(c(0,1,0,0,1,1))
fun <- benchmarkGeneratorQAP(6,3)
fun(c(1,0,1,1,0,0))
fun <- benchmarkGeneratorQAP(6,2,c(-1,1))
fun(c(1,0,1,1,0,0))
fun <- benchmarkGeneratorQAP(6,2,c(-1,1),g=matrix(runif(48),6))
fun(c(1,0,1,1,0,0))
fun(sample(c(0,1),6,TRUE))
```

**Description**

Create Quadratic Assignment Problem (QAP) Benchmark

**Usage**

```r
benchmarkGeneratorQAP(a, b)
```

**Arguments**

- `a` distance matrix
- `b` flow matrix

**Value**

the function of type cost=f(permutation)

**See Also**

`benchmarkGeneratorFSP`, `benchmarkGeneratorTSP`, `benchmarkGeneratorWT`

**Examples**

```r
set.seed(1)
n=5
# create a flow matrix
A <- matrix(0,n,n)
for(i in 1:n){
  for(j in i:n){
    if(i!=j){
      A[i,j] <- sample(100,1)
    }
  }
}
```
benchmarkGeneratorTSP

Create (Asymmetric) Travelling Salesperson Problem (TSP) Benchmark

Description

Creates a benchmark function for the (Asymmetric) Travelling Salesperson Problem. Path (Do not return to start of tour. Start and end of tour not fixed.) or Cycle (Return to start of tour). Symmetry depends on supplied distance matrix.

Usage

benchmarkGeneratorTSP(distanceMatrix, type = "Cycle")

Arguments

distanceMatrix  Matrix that collects the distances between travelled locations.
type  Can be "Cycle" (return to start, default) or "Path" (no return to start).

Value

the function of type cost=f(permutation)

See Also

benchmarkGeneratorQAP, benchmarkGeneratorFSP, benchmarkGeneratorWT

Examples

set.seed(1)
#create 5 random locations to be part of a tour
n=5
cities <- matrix(runif(2*n),,2)
#calculate distances between cities
cdist <- as.matrix(dist(cities))
#create objective functions (for path or cycle)

fun1 <- benchmarkGeneratorTSP(cdist, "Path")
fun2 <- benchmarkGeneratorTSP(cdist, "Cycle")
# evaluate
fun1(1:n)
fun1(n:1)
fun2(n:1)
fun2(1:n)

benchmarkGeneratorWT

Create single-machine total Weighted Tardiness (WT) Problem Benchmark

Description

 Creates a benchmark function for the single-machine total Weighted Tardiness Problem.

Usage

 benchmarkGeneratorWT(p, w, d)

Arguments

 p  processing times
 w  weights
 d  due dates

Value

 the function of type cost=f(permutation)

See Also

 benchmarkGeneratorQAP, benchmarkGeneratorTSP, benchmarkGeneratorFSP

Examples

 n=6  # processing times
 p <- sample(100,n,replace=TRUE)
 w <- sample(10,n,replace=TRUE)
 RDD <- c(0.2, 0.4, 0.6, 0.8, 1.0)
 TF <- c(0.2, 0.4, 0.6, 0.8, 1.0)
 i <- 1
 j <- 1
 P <- sum(p)
 d <- runif(n,min=P*(1-TF[i]-RDD[j]/2),max=P*(1-TF[i]+RDD[j]/2))
#create WT objective function
fun <- benchmarkGeneratorWT(p,w,d)
fun(1:n)
fun(n:1)

---

**correctionCNSD**  
**Correcting Conditional Negative Semi-Definiteness**

**Description**
Correcting, e.g., a distance matrix with chosen methods so that it becomes a CNSD matrix.

**Usage**
correctionCNSD(mat, method = "flip", tol = 1e-08)

**Arguments**
- mat: symmetric matrix, which should be at least of size 3x3
- method: string that specifies method for correction: spectrum clip "clip", spectrum flip "flip", nearest definite matrix "near", spectrum square "square", spectrum diffusion "diffusion".
- tol: tolerance value. Eigenvalues between -tol and tol are assumed to be zero.

**Value**
the corrected CNSD matrix

**References**

**See Also**
modelKriging

**Examples**
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x,distancePermutationInsert)
is.CNSD(D) #matrix should not be CNSD
D <- correctionCNSD(D)
is.CNSD(D) #matrix should now be CNSD
D
# note: to fix the negative distances, use repairConditionsDistanceMatrix.
# Or else, use correctionDistanceMatrix.
correctionDefinite  

Correcting Definiteness of a Matrix

Description
Correcting a (possibly indefinite) symmetric matrix with chosen approach so that it will have desired definiteness type: positive or negative semi-definite (PSD, NSD).

Usage
```
correctionDefinite(mat, type = "PSD", method = "flip", tol = 1e-08)
```

Arguments
- **mat**: symmetric matrix
- **type**: string that specifies type of correction: "PSD", "NSD" to enforce PSD or NSD matrices respectively.
- **method**: string that specifies method for correction: spectrum clip "clip", spectrum flip "flip", nearest definite matrix "near", spectrum square"square", spectrum diffusion "diffusion".
- **tol**: tolerance value. Eigenvalues between -tol and tol are assumed to be zero.

Value
- list with
  - **mat**: corrected matrix
  - **isIndefinite**: boolean, whether original matrix was indefinite
  - **lambda**: the eigenvalues of the original matrix
  - **lambdanew**: the eigenvalues of the corrected matrix
  - **U**: the matrix of eigenvectors
  - **a**: the transformation vector

References

See Also
- `modelKriging`
Examples

```r
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x,distancePermutationInsert)
is.NSD(D) #matrix should not be CNSD
D <- correctionDefinite(D,type="NSD")$mat
is.NSD(D) #matrix should now be CNSD
# different example: PSD kernel
D <- distanceMatrix(x,distancePermutationInsert)
K <- exp(-0.01*D)
is.PSD(K)
K <- correctionDefinite(K,type="PSD")$mat
is.PSD(K)
```

---

correctionDistanceMatrix

**Correction of a Distance Matrix**

**Description**

Convert (possibly non-euclidean or non-metric) distance matrix with chosen approach so that it becomes a CNSD matrix. Optionally, the resulting matrix is enforced to have positive elements and zero diagonal, with the repair parameter. Essentially, this is a combination of functions `correctionDefinite` or `correctionCNSD` with `repairConditionsDistanceMatrix`.

**Usage**

```r
correctionDistanceMatrix(
  mat,
  type = "NSD",
  method = "flip",
  repair = TRUE,
  tol = 1e-08
)
```

**Arguments**

- **mat**: symmetric distance matrix
- **type**: string that specifies type of correction: "CNSD","NSD" to enforce CNSD or NSD matrices respectively.
- **method**: string that specifies method for correction: spectrum clip "clip", spectrum flip "flip", nearest definite matrix "near", spectrum square"square", spectrum diffusion "diffusion", feature embedding "feature".
- **repair**: boolean, whether or not to use condition repair, so that elements are positive, and diagonal is zero.
- **tol**: tolerance value. Eigenvalues between -tol and tol are assumed to be zero.
correctionKernelMatrix

Value

list with corrected distance matrix mat, isCNSD (boolean, whether original matrix was CNSD) and transformation matrix A.

References


See Also
correctionDefinite, correctionCNSD, repairConditionsDistanceMatrix

Examples

```r
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x, distancePermutationInsert)
is.CNSD(D) # matrix should not be CNSD
D <- correctionDistanceMatrix(D)$mat
is.CNSD(D) # matrix should now be CNSD
```

correctionKernelMatrix

Correction of a Kernel (Correlation) Matrix

Description

Convert a non-PSD kernel matrix with chosen approach so that it becomes a PSD matrix. Optionally, the resulting matrix is enforced to have values between -1 and 1 and a diagonal of 1s, with the repair parameter. That means, it is (optionally) converted to a valid correlation matrix. Essentially, this is a combination of correctionDefinite with repairConditionsCorrelationMatrix.

Usage

correctionKernelMatrix(mat, method = "flip", repair = TRUE, tol = 1e-08)

Arguments

- **mat**: symmetric kernel matrix
- **method**: string that specifies method for correction: spectrum clip "clip", spectrum flip "flip", nearest definite matrix "near", spectrum square"square", spectrum diffusion "diffusion".
- **repair**: boolean, whether or not to use condition repair, so that elements between -1 and 1, and the diagonal values are 1.
- **tol**: tolerance value. Eigenvalues between -tol and tol are assumed to be zero.
createSimulatedTestFunction

**Value**

list with corrected kernel matrix `mat`, `isPSD` (boolean, whether original matrix was PSD), transformation matrix `A`, the matrix of eigenvectors (`U`) and the transformation vector (`a`)

**References**


**See Also**

correctionDefinite, repairConditionsCorrelationMatrix

**Examples**

```r
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x,distancePermutationInsert)
K <- exp(-0.01*D)
K
K <- correctionKernelMatrix(K)$mat
K
```

---

**createSimulatedTestFunction**

*Simulation-based Test Function Generator, Object Interface*

**Description**

Generate test functions for assessment of optimization algorithms with non-conditional or conditional simulation, based on real-world data. For a more streamlined interface, see `testFunctionGeneratorSim`.

**Usage**

```r
createSimulatedTestFunction(
  xsim,
  fit,
  nsim = 10,
  conditionalSimulation = TRUE,
  seed = NA
)
```

**Arguments**

- `xsim` list of samples in input space, for simulation
- `fit` an object generated by `modelKriging`
- `nsim` the number of simulations, or test functions, to be created
createSimulatedTestFunction

conditionalSimulation
  whether (TRUE) or not (FALSE) to use conditional simulation

seed
  a random number generator seed. Defaults to NA; which means no seed is set. For sake of reproducibility, set this to some integer value.

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

References

See Also
  modelKriging, simulate.modelKriging, testFunctionGeneratorSim

Examples
  nsim <- 10
  seed <- 12345
  n <- 6
  set.seed(seed)
  #target function:
  fun <- function(x){
    exp(-20*x) + sin(6*x^2) + x
  }
  # "vectorize" target
  f <- function(x){sapply(x,fun)}
  # distance function
  dF <- function(x,y)(sum((x-y)^2)) #sum of squares
  #start pdf creation
  # plot params
  par(mfrow=c(4,1),mar=c(2.3,2.5,0.2,0.2),mgp=c(1.4,0.5,0))
  #test samples for plots
  xtest <- as.list(seq(from=-0,by=0.005,to=1))
  plot(xtest,f(xtest),type="l",xlab="x",ylab="Obj. function")
  #evaluation samples (training)
  xb <- as.list(runif(n))
  yb <- f(xb)
  # support samples for simulation
  x <- as.list(sort(c(runif(100),unlist(xb))))
  # fit the model
  fit <- modelKriging(xb,yb,dF,control=list(}
distanceMatrix

Calculate Distance Matrix

Description

Calculate the distance between all samples in a list, and return as matrix.

Usage

distanceMatrix(X, distFun, ...)
**distanceNumericHamming**

**Value**

The distance matrix

**Examples**

```r
x <- list(5:1,c(2,4,5,1,3),c(5,4,3,1,2), sample(5))
distanceMatrix(x,distancePermutationHamming)
```

---

**distanceNumericHamming**

*Hamming Distance for Vectors*

**Description**

The number of unequal elements of two vectors (which may be of unequal length), divided by the number of elements (of the larger vector).

**Usage**

```r
distanceNumericHamming(x, y)
```

**Arguments**

- `x`: first vector
- `y`: second vector

**Value**

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1

**Examples**

#e.g., used for distance between bit strings
```r
x <- c(0,1,0,1,0)
y <- c(1,1,0,0,1)
distanceNumericHamming(x,y)
p <- replicate(10,sample(c(0,1),5,replace=TRUE),simplify=FALSE)
distanceMatrix(p,distanceNumericHamming)
```
distanceNumericLCStr  Longest Common Substring for Numeric Vectors

Description
Longest common substring distance for two numeric vectors, e.g., bit vectors.

Usage
distanceNumericLCStr(x, y)

Arguments
x  first vector (numeric)
y  second vector (numeric)

Value
cnumeric distance value  \(d(x, y)\)
, scaled to values between 0 and 1

Examples
#e.g., used for distance between bit strings
x <- c(0,1,0,1,0)
y <- c(1,1,0,0,1)
distanceNumericLCStr(x,y)
p <- replicate(10,sample(c(0,1),5,replace=TRUE),simplify=FALSE)
distanceMatrix(p,distanceNumericLCStr)

---

distanceNumericLevenshtein  Levenshtein Distance for Numeric Vectors

Description
Levenshtein distance for two numeric vectors, e.g., bit vectors.

Usage
distanceNumericLevenshtein(x, y)
Arguments

x  first vector (numeric)
y  second vector (numeric)

Value

numeric distance value \( d(x, y) \), scaled to values between 0 and 1

Examples

# e.g., used for distance between bit strings
x <- c(0,1,0,1,0)
y <- c(1,1,0,0,1)
distanceNumericLevenshtein(x,y)
p <- replicate(10,sample(c(0,1),5,replace=TRUE),simplify=FALSE)
distanceMatrix(p,distanceNumericLevenshtein)

---

distancePermutationAdjacency

Adjacency Distance for Permutations

Description

Bi-directional adjacency distance for permutations, depending on how often two elements are neighbours in both permutations x and y.

Usage

distancePermutationAdjacency(x, y)

Arguments

x  first permutation (integer vector)
y  second permutation (integer vector)

Value

numeric distance value \( d(x, y) \), scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)
distancePermutationChebyshev

References


Examples

```r
x <- 1:5
y <- 5:1
distancePermutationAdjacency(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationAdjacency)
```

---

distancePermutationChebyshev

*Chebyshev Distance for Permutations*

Description

Chebyshev distance for permutations. Specific to permutations is only the scaling to values of 0 to 1:

\[
d(x, y) = \frac{\max(|x - y|)}{(n - 1)}
\]

where \( n \) is the length of the permutations \( x \) and \( y \).

Usage

```r
distancePermutationChebyshev(x, y)
```

Arguments

- \( x \) first permutation (integer vector)
- \( y \) second permutation (integer vector)

Value

numeric distance value \( d(x, y) \), scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)
distancePermutationCos

Cosine Distance for Permutations

Description
The Cosine distance for permutations is derived from the Cosine similarity measure which has been applied in fields like text mining. It is based on the scalar product of two vectors (here: permutations).

Usage
distancePermutationCos(x, y)

Arguments
x first permutation (integer vector)
y second permutation (integer vector)

Value
numeric distance value

\[ d(x, y) \]
, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References

Examples
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationChebyshev(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationChebyshev)
Euclidean Distance for Permutations

distancePermutationEuclidean

Description

Euclidean distance for permutations, scaled to values between 0 and 1:

\[
d(x, y) = \frac{1}{r} \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}
\]

where \( n \) is the length of the permutations \( x \) and \( y \), and scaling factor \( r = \sqrt{2 \times 4 \times n \times (n + 1) \times (2 \times n + 1)/6} \) (if \( n \) is odd) or \( r = \sqrt{2 \times n \times (2 \times n - 1) \times (2 \times n + 1)/3} \) (if \( n \) is even).

Usage

distancePermutationEuclidean(x, y)

Arguments

\( x \quad \text{first permutation (integer vector)} \)
\( y \quad \text{second permutation (integer vector)} \)

Value

numeric distance value

\( d(x, y) \), scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

Examples

\( x \leftarrow 1:5 \)
\( y \leftarrow c(5,1,2,3,4) \)
\( \text{distancePermutationEuclidean}(x, y) \)
\( p \leftarrow \text{replicate}(10, \text{sample}(1:5), \text{simulate}=FALSE) \)
\( \text{distanceMatrix}(p, \text{distancePermutationEuclidean}) \)
distancePermutationHamming

Hamming Distance for Permutations

Description
Hamming distance for permutations, scaled to values between 0 and 1. That is, the number of unequal elements of two permutations, divided by the permutations length.

Usage
distancePermutationHamming(x, y)

Arguments
- x: first permutation (integer vector)
- y: second permutation (integer vector)

Value
numeric distance value
d(x, y)

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

Examples
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationHamming(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationHamming)

distancePermutationInsert

Insert Distance for Permutations

Description
The Insert Distance is an edit distance. It counts the minimum number of delete/insert operations required to transform one permutation into another. A delete/insert operation shifts one element to a new position. All other elements move accordingly to make place for the element. E.g., the following shows a single delete/insert move that sorts the corresponding permutation: 1 4 2 3 5 -> 1 2 3 4 5.
Usage

distancePermutationInsert(x, y)

Arguments

x first permutation (integer vector)
y second permutation (integer vector)

Value

numeric distance value $d(x, y)$

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References


Examples

```r
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationInsert(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationInsert)
```

---

distancePermutationInterchange

Interchange Distance for Permutations

Description

The interchange distance is an edit-distance, counting how many edit operation (here: interchanges, i.e., transposition of two arbitrary elements) have to be performed to transform permutation x into permutation y.

Usage

distancePermutationInterchange(x, y)
distancePermutationLCStr

Arguments

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)

Value

- numeric distance value
  \[ d(x, y) \]
  , scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References


Examples

```r
x <- 1:5
y <- c(1,4,3,2,5)
distancePermutationInterchange(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationInterchange)
```

---

**distancePermutationLCStr**

Longest Common Substring Distance for Permutations

Description

Distance of permutations. Based on the longest string of adjacent elements that two permutations have in common.

Usage

```r
distancePermutationLCStr(x, y)
```

Arguments

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)
  
  @return numeric distance value
  \[ d(x, y) \]
  , scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)
distancePermutationLee

Lee Distance for Permutations

Description

Usually a string distance, with slightly different definition. Adapted to permutations as:

\[ d(x, y) = \sum_{i=1}^{n} \min(|x_i - y_i|, n - |x_i - y_i|) \]

where \( n \) is the length of the permutations \( x \) and \( y \).

Usage

distancePermutationLee(x, y)

Arguments

- \( x \): first permutation (integer vector)
- \( y \): second permutation (integer vector)

Value

numeric distance value \( d(x, y) \), scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References

Examples
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationLee(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationLee)

---

distancePermutationLevenshtein
Levenshtein Distance for Permutations

Description
Levenshtein Distance, often just called "Edit Distance". The number of insertions, substitutions or deletions to turn one permutation (or string of equal length) into another.

Usage
distancePermutationLevenshtein(x, y)

Arguments
x first permutation (integer vector)
y second permutation (integer vector)

Value
numeric distance value $d(x, y)$, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References

Examples
x <- 1:5
y <- c(1,2,5,4,3)
distancePermutationLevenshtein(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationLevenshtein)
**distancePermutationLex**

*Lexicographic permutation distance*

**Description**

This function calculates the lexicographic permutation distance. That is the difference of positions that both positions would receive in a lexicographic ordering. Note, that this distance measure can quickly become inaccurate if the length of the permutations grows too large, due to being based on the factorial of the length. In general, permutations longer than 100 elements should be avoided.

**Usage**

`distancePermutationLex(x, y)`

**Arguments**

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)

**Value**

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**See Also**

`lexicographicPermutationOrderNumber`

**Examples**

```r
x <- 1:5
y <- c(1,2,3,5,4)
distancePermutationLex(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationLex)
```
distancePermutationManhattan

Manhattan Distance for Permutations

Description

Manhattan distance for permutations, scaled to values between 0 and 1:

\[
d(x, y) = \frac{1}{n} \sum_{i=1}^{n} |x_i - y_i|
\]

where \( n \) is the length of the permutations \( x \) and \( y \), and scaling factor \( r = \frac{(n^2 - 1)}{2} \) (if \( n \) is odd) or \( r = \frac{((n^2)}/2 \) (if \( n \) is even).

Usage

\[
distancePermutationManhattan(x, y)
\]

Arguments

- \( x \) first permutation (integer vector)
- \( y \) second permutation (integer vector)

Value

numeric distance value

\( d(x, y) \), scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

Examples

\[
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationManhattan(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationManhattan)
\]
distancePermutationPosition

Position Distance for Permutations

Description

Position distance (or Spearman's Correlation Coefficient), scaled to values between 0 and 1.

Usage

distancePermutationPosition(x, y)

Arguments

x first permutation (integer vector)
y second permutation (integer vector)

Value

numeric distance value

\[ d(x, y) \]
, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

References


Examples

```r
x <- 1:5
y <- c(1,3,5,4,2)
distancePermutationPosition(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationPosition)
```
distancePermutationPosition2

*Squared Position Distance for Permutations*

**Description**

Squared position distance (or Spearman's Footrule), scaled to values between 0 and 1.

**Usage**

```
distancePermutationPosition2(x, y)
```

**Arguments**

- `x` first permutation (integer vector)
- `y` second permutation (integer vector)

**Value**

numeric distance value

\[ d(x, y) \]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**References**


**Examples**

```r
x <- 1:5
y <- c(1,3,5,4,2)
distancePermutationPosition2(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationPosition2)
```
**distancePermutationR**  
*R-Distance for Permutations*

**Description**

R distance or unidirectional adjacency distance. Based on count of number of times that a two element sequence in x also occurs in y, in the same order.

**Usage**

\[
\text{distancePermutationR}(x, y)
\]

**Arguments**

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)

**Value**

numeric distance value

\[
d(x, y)
\]

, scaled to values between 0 and 1 (based on the maximum possible distance between two permutations)

**References**


**Examples**

```r
x <- 1:5
y <- c(1,2,3,5,4)
distancePermutationR(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distanceMatrix(p, distancePermutationR)
```
**distancePermutationSwap**

*Swap-Distance for Permutations*

**Description**

The swap distance is an edit-distance, counting how many edit operation (here: swaps, i.e., transposition of two adjacent elements) have to be performed to transform permutation $x$ into permutation $y$.

**Usage**

```r
distancePermutationSwap(x, y)
```

**Arguments**

- **x**: first permutation (integer vector)
- **y**: second permutation (integer vector)

**Value**

numeric distance value $d(x, y)$, scaled to values between $0$ and $1$ (based on the maximum possible distance between two permutations)

**References**


**Examples**

```r
x <- 1:5
y <- c(1,2,3,5,4)
distancePermutationSwap(x,y)
p <- replicate(10,sample(1:5),simplify=FALSE)
distanceMatrix(p,distancePermutationSwap)
```
**distanceRealEuclidean  Euclidean Distance**

**Description**

The Euclidean distance for real vectors.

**Usage**

\[ \text{distanceRealEuclidean}(x, y) \]

**Arguments**

- \( x \): first real vector
- \( y \): second real vector

**Value**

numeric distance value

\[ d(x, y) \]

**Examples**

```r
x <- runif(5)
y <- runif(5)
distanceRealEuclidean(x, y)
```

---

**distanceStringHamming  Hamming Distance for Strings**

**Description**

Number of unequal letters in two strings.

**Usage**

\[ \text{distanceStringHamming}(x, y) \]

**Arguments**

- \( x \): first string (class: character)
- \( y \): second string (class: character)
distanceStringLCstr

Value
numeric distance value
\[ d(x, y) \]

Examples

distanceStringHamming("ABCD","AACC")

distanceStringLCstr("ABC","AACC")

Description
Distance between strings, based on the longest common substring.

Usage
distanceStringLCstr(x, y)

Arguments
\begin{itemize}
\item x \text{ first string (class: character)}
\item y \text{ second string (class: character)}
\end{itemize}

Value
numeric distance value
\[ d(x, y) \]

Examples
distanceStringLCstr("ABC","AACC")
### distanceStringLevenshtein

*Levenshtein Distance for Strings*

**Description**
Number of insertions, deletions and substitutions to transform one string into another

**Usage**
distanceStringLevenshtein(x, y)

**Arguments**
x first string (class: character)
y second string (class: character)

**Value**
numeric distance value 
\[ d(x, y) \]

**Examples**
distanceStringLevenshtein("ABCD", "AACC")

---

### distanceVector

*Calculate Distance Vector*

**Description**
Calculate the distance between a single sample and all samples in a list.

**Usage**
distanceVector(a, X, distFun, ...)

**Arguments**
a A single sample which is a suitable input for distFun
X list of samples, where each list element is a suitable input for distFun
distFun Distance function of type f(x,y)=r, where r is a scalar and x and y are elements whose distance is evaluated.
... further arguments passed to distFun
infillExpectedImprovement

Value

A numerical vector of distances

Examples

```r
x <- 1:5
y <- list(5:1,c(2,4,5,1,3),c(5,4,3,1,2))
distanceVector(x,y,distancePermutationHamming)
```

Description

This function calculates the Expected Improvement of candidate solutions, based on predicted means, standard deviations (uncertainty) and the best known objective function value so far.

Usage

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

Arguments

- `mean`: predicted mean values
- `sd`: predicted standard deviation
- `min`: minimum of all observations so far

Value

Returns the negative logarithm of the Expected Improvement.

is.CNSD

Check for Conditional Negative Semi-Definiteness

Description

This function checks whether a symmetric matrix is Conditionally Negative Semi-Definite (CNSD). Note that this function does not check whether the matrix is actually symmetric.

Usage

```r
is.CNSD(X, method = "alg1", tol = 1e-08)
```
Arguments

- **X**: a symmetric matrix
- **method**: a string, specifying the method to be used. "alg1" is based on algorithm 1 in Ikramov and Savel’eva (2000). "alg2" is based on theorem 3.2 in Ikramov and Savel’eva (2000). "eucl" is based on Glunt (1990).
- **tol**: tolerance value. Eigenvalues between -tol and tol are assumed to be zero.

Symmetric, CNSD matrices are, e.g., euclidean distance matrices, which are required to produce Positive Semi-Definite correlation or kernel matrices. Such matrices are used in models like Kriging or Support Vector Machines.

Value

boolean, which is TRUE if X is CNSD

References


See Also

- is.NSD, is.PSD

Examples

# The following permutations will produce
# a non-CNSD distance matrix with Insert distance
# and a CNSD distance matrix with Hamming distance
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x,distancePermutationInsert)
is.CNSD(D,"alg1")
is.CNSD(D,"alg2")
is.CNSD(D,"eucl")
D <- distanceMatrix(x,distancePermutationHamming)
is.CNSD(D,"alg1")
is.CNSD(D,"alg2")
is.CNSD(D,"eucl")
is.NSD

Check for Negative Semi-Definiteness

Description

This function checks whether a symmetric matrix is Negative Semi-Definite (NSD). That means, it is determined whether all eigenvalues of the matrix are non-positive. Note that this function does not check whether the matrix is actually symmetric.

Usage

is.NSD(X, tol = 1e-08)

Arguments

X
  a symmetric matrix

tol
  tolerance value. Eigenvalues between \(-\text{tol}\) and \(\text{tol}\) are assumed to be zero.

Symmetric, NSD matrices are, e.g., correlation or kernel matrices. Such matrices are used in models like Kriging or Support Vector regression.

Value

boolean, which is TRUE if X is NSD

See Also

is.CNSD, is.PSD

Examples

# The following permutations will produce
# a non-PSD kernel matrix with Insert distance
# and a PSD distance matrix with Hamming distance
# (for the given theta value of 0.01)-
# The respective negative should be (non-) NSD
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
K <- exp(-0.01*distanceMatrix(x,distancePermutationInsert))
is.NSD(-K)
K <- exp(-0.01*distanceMatrix(x,distancePermutationHamming))
is.NSD(-K)
is.PSD  

Check for Positive Semi-Definiteness

Description

This function checks whether a symmetric matrix is Positive Semi-Definite (PSD). That means, it is determined whether all eigenvalues of the matrix are non-negative. Note that this function does not check whether the matrix is actually symmetric.

Usage

is.PSD(X, tol = 1e-08)

Arguments

X  
a symmetric matrix

tol  
tolerance value. Eigenvalues between -tol and tol are assumed to be zero.

Symmetric, PSD matrices are, e.g., correlation or kernel matrices. Such matrices are used in models like Kriging or Support Vector regression.

Value

boolean, which is TRUE if X is PSD

See Also

is.CNSD, is.NSD

Examples

# The following permutations will produce
# a non-PSD kernel matrix with Insert distance
# and a PSD distance matrix with Hamming distance
# (for the given theta value of 0.01)
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
K <- exp(-0.01*distanceMatrix(x,distancePermutationInsert))
is.PSD(K)
K <- exp(-0.01*distanceMatrix(x,distancePermutationHamming))
is.PSD(K)
**kernelMatrix**

*Calculate Kernel Matrix*

**Description**

Calculate the similarities between all samples in a list, and return as matrix.

**Usage**

`kernelMatrix(X, kernFun, ...)`

**Arguments**

- `X` list of samples, where each list element is a suitable input for `kernFun`
- `kernFun` Kernel function of type \( f(x,y) = r \), where \( r \) is a scalar and \( x \) and \( y \) are elements whose similarity is evaluated.
- `...` further arguments passed to `distFun`

**Value**

The similarity / kernel matrix

**Examples**

```r
x <- list(5:1,c(2,4,5,1,3),c(5,4,3,1,2), sample(5))
kernFun <- function(x,y){
  exp(-distancePermutationHamming(x,y))
}
kernelMatrix(x,distancePermutationHamming)
```

**landscapeGeneratorGaussian**

*Create Gaussian Landscape*

**Description**

This function is loosely based on the Gaussian Landscape Generator by Bo Yuan and Marcus Gallagher. It creates a Gaussian Landscape every time it is called. This Landscape can be evaluated like a function. To adapt to combinatorial spaces, the Gaussians are here based on a user-specified distance measure. Due to the expected nature of combinatorial spaces and their lack of direction, the resulting Gaussians are much simplified in comparison to the continuous, vector-valued case (e.g., no rotation). Since the CEGO package is tailored to minimization, the landscape is inverted.
landscapeGeneratorGaussian

Usage

landscapeGeneratorGaussian(
    nGaussian = 10,
    theta = 1,
    ratio = 0.2,
    seed = 1,
    distanceFunction,
    creationFunction
)

Arguments

nGaussian number of Gaussian components in the landscape. Default is 10.
theta controls width of Gaussian components as a multiplier. Default is 1.
ratio minimal function value of the local minima. Default is 0.2. (Note: Global
minimum will be at zero, local minima will be in range [ratio;1])
seed seed for the random number generator used before creation of the landscape.
Generator status will be saved and reset afterwards.
distanceFunction A function of type \( f(x,y) \), to evaluate distance between to samples in their
given representation.
creationFunction function to randomly generate the centers of the Gaussians, in form of their
given representation.

Value

returns a function. The function requires a list of candidate solutions as its input, where each solution
is suitable for use with the distance function.

References

B. Yuan and M. Gallagher (2003) "On Building a Principled Framework for Evaluating and Test-
ing Evolutionary Algorithms: A Continuous Landscape Generator". In Proceedings of the 2003
Congress on Evolutionary Computation, IEEE, pp. 451-458, Canberra, Australia.

Examples

#rng seed
seed=101
# distance function
dF <- function(x,y)(sum((x-y)^2)) #sum of squares
#dF <- function(x,y)sqrt(sum((x-y)^2)) #euclidean distance
# creation function
cF <- function()runif(1)
# plot pars
par(mfrow=c(3,1),mar=c(3.5,3.5,0.2,0.2),mgp=c(2,1,0))
## uni modal distance landscape
# set seed


### landscapeGeneratorMUL

**Multimodal Fitness Landscape**

**Description**

This function generates multi-modal fitness landscapes based on distance measures. The fitness is the minimal distance to several reference individuals or centers. Hence, each reference individual is an optimum of the landscape.

**Usage**

```r
landscapeGeneratorMUL(ref, distanceFunction)
```

**Arguments**

- `ref` list of reference individuals / centers
- `distanceFunction` Distance function, used to evaluate \( d(x, ref[n]) \), where \( x \) is an arbitrary new individual

**Value**

returns a function. The function requires a list of candidate solutions as its input, where each solution is suitable for use with the distance function. The function returns a numeric vector.

**See Also**

`landscapeGeneratorUNI`, `landscapeGeneratorGaussian`
Examples

fun <- landscapeGeneratorMUL(ref=list(1:7,c(2,4,1,5,3,7,6)),distancePermutationCos)
x <- 1:7
fun(list(x))
x <- c(2,4,1,5,3,7,6)
fun(list(x))
x <- 7:1
fun(list(x))
x <- sample(7)
fun(list(x))
## multiple solutions at once:
x <- append(list(1:7,c(2,4,1,5,3,7,6)),replicate(5,sample(7),FALSE))
fun(x)

landscapeGeneratorUNI  Unimodal Fitness Landscape

Description

This function generates uni-modal fitness landscapes based on distance measures. The fitness is the
distance to a reference individual or center. Hence, the reference individual is the optimum of the
landscape. This function is essentially a wrapper for the landscapeGeneratorMUL

Usage

landscapeGeneratorUNI(ref, distanceFunction)

Arguments

ref            reference individual
distanceFunction            Distance function, used to evaluate d(x, ref), where x is an arbitrary new individ-

Value

returns a function. The function requires a list of candidate solutions as its input, where each
solution is suitable for use with the distance function. The function returns a numeric vector.

References

Moraglio, Alberto, Yong-Hyuk Kim, and Yourim Yoon. "Geometric surrogate-based optimisation
for permutation-based problems." Proceedings of the 13th annual conference companion on Genetic
and evolutionary computation. ACM, 2011.

See Also

landscapeGeneratorMUL, landscapeGeneratorGaussian
Examples

```r
fun <- landscapeGeneratorUNI(ref=1:7,distancePermutationCos)
## for single solutions, note that the function still requires list input:
x <- 1:7
fun(list(x))
x <- 7:1
fun(list(x))
x <- sample(7)
fun(list(x))
## multiple solutions at once:
x <- replicate(5,sample(7),FALSE)
fun(x)
```

---

**lexicographicPermutationOrderNumber**

*Lexicographic order number*

Description

This function returns the position-number that a permutation would receive in a lexicographic ordering. It is used in the lexicographic distance measure.

Usage

```r
lexicographicPermutationOrderNumber(x)
```

Arguments

- `x`: permutation (integer vector)

Value

numeric value giving position in lexicographic order.

See Also

`distancePermutationLex`

Examples

```r
lexicographicPermutationOrderNumber(1:5)
lexicographicPermutationOrderNumber(c(1,2,3,5,4))
lexicographicPermutationOrderNumber(c(1,2,4,3,5))
lexicographicPermutationOrderNumber(c(1,2,4,5,3))
lexicographicPermutationOrderNumber(c(1,2,5,3,4))
lexicographicPermutationOrderNumber(c(1,2,5,4,3))
lexicographicPermutationOrderNumber(c(1,3,2,4,5))
lexicographicPermutationOrderNumber(5:1)
```
modelKriging

Kriging Model

Description

Implementation of a distance-based Kriging model, e.g., for mixed or combinatorial input spaces. It is based on employing suitable distance measures for the samples in input space.

Usage

modelKriging(x, y, distanceFunction, control = list())

Arguments

x
list of samples in input space

y
column vector of observations for each sample

distanceFunction
a suitable distance function of type f(x1,x2), returning a scalar distance value, preferably between 0 and 1. Maximum distances larger 1 are no problem, but may yield scaling bias when different measures are compared. Should be non-negative and symmetric. It can also be a list of several distance functions. In this case, Maximum Likelihood Estimation (MLE) is used to determine the most suited distance measure. The distance function may have additional parameters. For that case, see distanceParametersLower/Upper in the controls. If distanceFunction is missing, it can also be provided in the control list.

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

lower lower boundary for theta, default is 1e-6
upper upper boundary for theta, default is 100
corr function to be used for correlation modelling, default is fcorrGauss
algTheta algorithm used to find theta (as well as p and lambda), default is optimInterface.
algThetaControl list of controls passed to algTheta.
useLambda whether or not to use the regularization constant lambda (nugget effect). Default is FALSE.
lambdaLower lower boundary for lambda (log scale), default is -6
lambdaUpper upper boundary for lambda (log scale), default is 0
distanceParametersLower lower boundary for parameters of the distance function, default is NA which means there are no distance function parameters. If several distance functions are supplied, this should be a list of lower boundary vectors for each function.
\text{distanceParametersUpper} \quad \text{upper boundary for parameters of the distance function, default is NA which means there are no distance function parameters. If several distance functions are supplied, this should be a list of upper boundary vectors for each function.}

\text{distances} \quad \text{a distance matrix. If available, this matrix is used for model building, instead of calculating the distance matrix using the parameters distanceFunction. Default is NULL.}

\text{scaling} \quad \text{If TRUE: Distances values are divided by maximum distance to avoid scale bias.}

\text{reinterpolate} \quad \text{If TRUE: reinterpolation is used to generate better uncertainty estimates in the presence of noise.}

\text{combineDistances} \quad \text{By default, several distance functions or matrices are subject to a likelihood based decision, choosing one. If this parameter is TRUE, they are instead combined by determining a weighted sum. The weighting parameters are determined by MLE.}

\text{userParameters} \quad \text{By default: (NULL). Else, this vector is used instead of MLE to specify the model parameters, in the following order: kernel parameters, distance weights, lambda, distance parameters.}

\text{indefiniteMethod} \quad \text{The specific method used for correction: spectrum "clip", spectrum "flip", spectrum "square", spectrum "diffusion", feature embedding "feature", nearest definite matrix "near". Default is no correction: "none". See Zaefferer and Bartz-Beielstein (2016).}

\text{indefiniteType} \quad \text{The general type of correction for indefiniteness: "NSD","CNSD" or the default "PSD". See Zaefferer and Bartz-Beielstein (2016). Note, that feature embedding may not work in case of multiple distance functions.}

\text{indefiniteRepair} \quad \text{boolean, whether conditions of the distance matrix (in case of "NSD","CNSD" correction type) or correlation matrix (in case of "PSD" correction type) are repaired.}

\text{Details}

The basic Kriging implementation is based on the work of Forrester et al. (2008). For adaptation of Kriging to mixed or combinatorial spaces, as well as choosing distance measures with Maximum Likelihood Estimation, see the other two references (Zaefferer et al., 2014).

\text{Value}

an object of class \text{modelKriging} containing the options (see control parameter) and determined parameters for the model:

\text{theta} \quad \text{parameters of the kernel / correlation function determined with MLE.}
\text{lambda} \quad \text{regularization constant (nugget) lambda}
\text{yMu} \quad \text{vector of observations y, minus MLE of mu}
\text{SSQ} \quad \text{Maximum Likelihood Estimate (MLE) of model parameter sigma^2}
\text{mu} \quad \text{MLE of model parameter mu}
\text{Psi} \quad \text{correlation matrix Psi}
Psinv  inverse of Psi
nevals  number of Likelihood evaluations during MLE of theta/lambda/p
distanceFunctionIndexMLE  If a list of several distance measures (distanceFunction) was given,
                      this parameter contains the index value of the measure chosen with MLE.

References


See Also

  predict.modelKriging

Examples

```r
# Set random number generator seed
set.seed(1)

# Simple test landscape
fn <- landscapeGeneratorUNI(1:5,distancePermutationHamming)

# Generate data for training and test
x <- unique(replicate(40,sample(5),FALSE))
xtest <- x[-(1:15)]
x <- x[1:15]

# Determine true objective function values
y <- fn(x)
ytest <- fn(xtest)

# Build model
fit <- modelKriging(x,y,distancePermutationHamming,
t       control=list(algThetaControl=list(method="L-BFGS-B"),useLambda=FALSE))

# Predicted obj. function values
ypred <- predict(fit,xtest)$y

# Uncertainty estimate
fit$s predAll <- TRUE
spred <- predict(fit,xtest)$s

# Plot
plot(ytest,ypred,xlab="true value",ylab="predicted value",
pch=20,xlim=c(0.3,1),ylim=c(min(ypred)-0.1,max(ypred)+0.1))
segments(ytest, ypred-spred,ytest, ypred+spred)
espsilon = 0.02
segments(ytest-epsilon,ypred-spred,ytest+epsilon,ypred-spred)
```
segments(ytest-epsilon,ypred+spred,ytest+epsilon,ypred+spred)
abline(0,1,lty=2)
# Use a different/custom optimizer (here: SANN) for maximum likelihood estimation:
# (Note: Bound constraints are recommended, to avoid Inf values.
# This is really just a demonstration. SANN does not respect bound constraints.)
optimizer1 <- function(x,fun,lower=NULL,upper=NULL,control=NULL,...){
    res <- optim(x,fun,method="SANN",control=list(maxit=100),...)
    list(xbest=res$par,ybest=res$value,count=res$counts)
}
fit <- modelKriging(x,y,distancePermutationHamming,
                     control=list(algTheta=optimizer1,useLambda=FALSE))
#One-dimensional optimizer (Brent). Note, that Brent will not work when
#several parameters have to be set, e.g., when using nugget effect (lambda).
#However, Brent may be quite efficient otherwise.
optimizer2 <- function(x,fun,lower,upper,control=NULL,...){
    res <- optim(x,fun,method="Brent",lower=lower,upper=upper,...)
    list(xbest=res$par,ybest=res$value,count=res$counts)
}
fit <- modelKriging(x,y,distancePermutationHamming,
                     control=list(algTheta=optimizer2,useLambda=FALSE))

modellinear

Distance based Linear Model

Description

A simple linear model based on arbitrary distances. Comparable to a k nearest neighbor model, but potentially able to extrapolate into regions of improvement. Used as a simple baseline by Zaefferer et al.(2014).

Usage

modellinear(x, y, distanceFunction, control = list())

Arguments

x
  list of samples in input space

y
  matrix, vector of observations for each sample

distanceFunction
  a suitable distance function of type f(x1,x2), returning a scalar distance value, preferably between 0 and 1. Maximum distances larger 1 are no problem, but may yield scaling bias when different measures are compared. Should be non-negative and symmetric.

control
  currently unused, defaults to list()
**Value**

a fit (list, `modelLinear`), with the options and found parameters for the model which has to be passed to the predictor function:

- `x` samples in input space (see parameters)
- `y` observations for each sample (see parameters)
- `distanceFunction` distance function (see parameters)

**References**


**See Also**

- `predict.modelLinear`

**Examples**

```r
#set random number generator seed
set.seed(1)
#simple test landscape
fn <- landscapeGeneratorUNI(1:5,distancePermutationHamming)
#generate data for training and test
x <- unique(replicate(40,sample(5),FALSE))
x <- x[-(1:15)]
xtest <- x[-(1:15)]
#determine true objective function values
y <- fn(x)
ytest <- fn(xtest)
#build model
fit <- modelLinear(x,y,distancePermutationHamming)
#predicted obj. function values
ypred <- predict(fit,xtest)$y
#plot
plot(ytest,ypred,xlab="true value",ylab="predicted value",
pch=20,xlim=c(0.3,1),ylim=c(min(ypred)-0.1,max(ypred)+0.1))
abline(0,1,lty=2)
```

---

**Description**

Implementation of a distance-based Radial Basis Function Network (RBFN) model, e.g., for mixed or combinatorial input spaces. It is based on employing suitable distance measures for the samples in input space. For reference, see the paper by Moraglio and Kattan (2011).
Usage

modelRBFN(x, y, distanceFunction, control = list())

Arguments

x list of samples in input space
y column vector of observations for each sample
distanceFunction
  a suitable distance function of type f(x1,x2), returning a scalar distance value,
  preferably between 0 and 1. Maximum distances larger 1 are no problem, but
  may yield scaling bias when different measures are compared. Should be non-
  negative and symmetric.
control (list), with the options for the model building procedure:
  beta Parameter of the radial basis function: exp(-beta*D), where D is the dis-
  tance matrix. If beta is not specified, the heuristic in fbeta will be used to
  determine it, which is default behavior.
  fbeta Function f(x) to calculate the beta parameter, x is the maximum distance
  observed in the input data. Default function is 1/(2*(x^2)).
  distances a distance matrix. If available, this matrix is used for model build-
  ing, instead of calculating the distance matrix using the parameters distanceFunction.
  Default is NULL.

Value

a fit (list, modelRBFN), with the options and found parameters for the model which has to be passed
  to the predictor function:
SSQ  Variance of the observations (y)
centers Centers of the RBFN model, samples in input space (see parameters)
w  Model parameters (weights) w
Phi  Gram matrix
Phinv (Pseudo)-Inverse of Gram matrix
w0  Mean of observations (y)
dMax Maximum observed distance
D Matrix of distances between all samples
beta See parameters
fbeta See parameters
distanceFunction See parameters

References

Moraglio, Alberto, and Ahmed Kattan. “Geometric generalisation of surrogate model based op-
  timisation to combinatorial spaces.” Evolutionary Computation in Combinatorial Optimiztion.
See Also

`predict.modelRBFN`

Examples

```r
#set random number generator seed
set.seed(1)
#simple test landscape
fn <- landscapeGeneratorUNI(1:5,distancePermutationHamming)
#generate data for training and test
x <- unique(replicate(40,sample(5),FALSE))
xtest <- x[-(1:15)]
x <- x[1:15]
#determin true objective function values
y <- fn(x)
ytest <- fn(xtest)
#build model
fit <- modelRBFN(x,y,distancePermutationHamming)
#predicted obj. function values
ypred <- predict(fit,xtest)$y
#plot
plot(ytest,ypred,xlab="true value",ylab="predicted value",
     pch=20,xlim=c(0.3,1),ylim=c(min(ypred)-0.1,max(ypred)+0.1))
abline(0,1,lty=2)
```

mutationBinaryBitFlip  Bit-flip Mutation for Bit-strings

Description

Given a population of bit-strings, this function mutates all individuals by randomly inverting one or more bits in each individual.

Usage

`mutationBinaryBitFlip(population, parameters)`

Arguments

- `population`: List of bit-strings
- `parameters`: list of parameters: `parameters$mutationRate` => mutation rate, specifying number of bits flipped. Should be in range between zero and one

Value

mutated population
**mutationBinaryBlockInversion**

*Block Inversion Mutation for Bit-strings*

**Description**
Given a population of bit-strings, this function mutates all individuals by inverting a whole block, randomly selected.

**Usage**
mutationBinaryBlockInversion(population, parameters)

**Arguments**
- **population**: List of bit-strings
- **parameters**: list of parameters: parameters$mutationRate => mutation rate, specifying number of bits flipped. Should be in range between zero and one

**Value**
mutated population

---

**mutationBinaryCycle**

*Cycle Mutation for Bit-strings*

**Description**
Given a population of bit-strings, this function mutates all individuals by cyclical shifting the string to the right or left.

**Usage**
mutationBinaryCycle(population, parameters)

**Arguments**
- **population**: List of bit-strings
- **parameters**: list of parameters: parameters$mutationRate => mutation rate, specifying number of bits flipped. Should be in range between zero and one

**Value**
mutated population
**mutationBinarySingleBitFlip**

*Single Bit-flip Mutation for Bit-strings*

**Description**

Given a population of bit-strings, this function mutates all individuals by randomly inverting one bit in each individual. Due to the fixed mutation rate, this is computationally faster.

**Usage**

`mutationBinarySingleBitFlip(population, parameters)`

**Arguments**

- `population`: List of bit-strings
- `parameters`: not used

**Value**

mutated population

---

**mutationPermutationInsert**

*Insert Mutation for Permutations*

**Description**

Given a population of permutations, this function mutates all individuals by randomly selecting two indices. The element at index1 is moved to position index2, other elements

**Usage**

`mutationPermutationInsert(population, parameters = list())`

**Arguments**

- `population`: List of permutations
- `parameters`: list of parameters, currently only uses parameters$mutationRate, which should be between 0 and 1 (but can be larger than 1). The mutation rate determines the number of reversals performed, relative to the permutation length (N). 0 means none. 1 means N reversals. The default is 1/N.

**Value**

mutated population
**mutationPermutationInterchange**

*Interchange Mutation for Permutations*

**Description**

Given a population of permutations, this function mutates all individuals by randomly interchanging two arbitrary elements of the permutation.

**Usage**

```r
mutationPermutationInterchange(population, parameters = list())
```

**Arguments**

- **population**: List of permutations
- **parameters**: list of parameters, currently only uses parameters$mutationRate, which should be between 0 and 1 (but can be larger than 1). The mutation rate determines the number of interchanges performed, relative to the permutation length (N). 0 means none. 1 means N interchanges. The default is 1/N.

**Value**

mutated population

---

**mutationPermutationReversal**

*Reversal Mutation for Permutations*

**Description**

Given a population of permutations, this function mutates all individuals by randomly selecting two indices, and reversing the respective sub-permutation.

**Usage**

```r
mutationPermutationReversal(population, parameters = list())
```

**Arguments**

- **population**: List of permutations
- **parameters**: list of parameters, currently only uses parameters$mutationRate, which should be between 0 and 1 (but can be larger than 1). The mutation rate determines the number of reversals performed, relative to the permutation length (N). 0 means none. 1 means N reversals. The default is 1/N.
mutationPermutationSwap

*Swap Mutation for Permutations*

**Description**

Given a population of permutations, this function mutates all individuals by randomly interchanging two adjacent elements of the permutation.

**Usage**

```r
mutationPermutationSwap(population, parameters = list())
```

**Arguments**

- `population` List of permutations
- `parameters` list of parameters, currently only uses `parameters$mutationRate`, which should be between 0 and 1 (but can be larger than 1). The mutation rate determines the number of swaps performed, relative to the permutation length (N). 0 means none. 1 means N swaps. The default is 1/N.

**Value**

mutated population

mutationSelfAdapt

*Self-adaptive mutation operator*

**Description**

This mutation function selects an operator and mutationRate (provided in `parameters$mutationFunctions`) based on self-adaptive parameters chosen for each individual separately.

**Usage**

```r
mutationSelfAdapt(population, parameters)
```

**Arguments**

- `population` List of permutations
- `parameters` list, contains the available single mutation functions (mutationFunctions), and a data.frame that collects the chosen function and mutation rate for each individual (selfAdapt).
mutationStringRandomChange

Mutation for Strings

Description

Given a population of strings, this function mutates all individuals by randomly changing an element of the string.

Usage

mutationStringRandomChange(population, parameters = list())
Arguments

- **population**: List of permutations
- **parameters**: list of parameters, with parameters$mutationRate and parameters$lts. parameters$mutationRate should be between 0 and 1 (but can be larger than 1). The mutation rate determines the number of interchanges performed, relative to the permutation length (N). 0 means none. 1 means N interchanges. The default is 1/N. parameters$lts are the possible letters in the string.

Value

mutated population

nearCNSD | Nearest CNSD matrix

Description

This function implements the alternating projection algorithm by Glunt et al. (1990) to calculate the nearest conditionally negative semi-definite (CNSD) matrix (or: the nearest Euclidean distance matrix). The function is similar to the nearPD function from the Matrix package, which implements a very similar algorithm for finding the nearest Positive Semi-Definite (PSD) matrix.

Usage

```r
nearCNSD(
  x,
  eig.tol = 1e-08,
  conv.tol = 1e-08,
  maxit = 1000,
  conv.norm.type = "F"
)
```

Arguments

- **x**: symmetric matrix, to be turned into a CNSD matrix.
- **eig.tol**: eigenvalue tolerance value. Eigenvalues between -tol and tol are assumed to be zero.
- **conv.tol**: convergence tolerance value. The algorithm stops if the norm of the difference between two iterations is below this value.
- **maxit**: maximum number of iterations. The algorithm stops if this value is exceeded, even if not converged.
- **conv.norm.type**: type of norm, by default the F-norm (Frobenius). See norm for other choices.
Value

- **mat**: nearest CNSD matrix
- **normF**: F-norm between original and resulting matrices
- **iterations**: the number of performed
- **rel.tol**: the relative value used for the tolerance convergence criterion
- **converged**: a boolean that records whether the algorithm

References


See Also

nearPD, correctionCNSD, correctionDistanceMatrix

Examples

```r
# example using Insert distance with permutations:
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x,distancePermutationInsert)
print(D)
is.CNSD(D)
nearD <- nearCNSD(D)
print(nearD)
is.CNSD(nearD$mat)
# or example matrix from Glunt et al. (1990):
D <- matrix(c(0,1,1,1,0,9,1,9,0),3,3)
print(D)
is.CNSD(D)
nearD <- nearCNSD(D)
print(nearD)
is.CNSD(nearD$mat)
# note, that the resulting values given by Glunt et al. (1990) are 19/9 and 76/9
```

---

**optim2Opt**

Two-Opt

Description

Implementation of a Two-Opt local search.

Usage

```r
optim2Opt(x = NULL, fun, control = list())
```
Arguments

x  start solution of the local search
fun  function that determines cost or length of a route/permutation
control  (list), with the options:
  archive  Whether to keep all candidate solutions and their fitness in an archive (TRUE) or not (FALSE). Default is TRUE.
  budget  The limit on number of target function evaluations (stopping criterion) (default: 100)
  creationFunction  Function to create individuals/solutions in search space. Default is a function that creates random permutations of length 6
  vectorized  Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE

Value

a list with:

xbest  best solution found
ybest  fitness of the best solution
count  number of performed target function evaluations

References


See Also

optimCEGO, optimEA, optimRS, optimMaxMinDist

Examples

seed=0
#distance
dF <- distancePermutationHamming
#creation
cF <- function()sample(5)
#objective function
lF <- landscapeGeneratorUNI(1:5,dF)
#start optimization
set.seed(seed)
res <- optim2Opt(lF,list(creationFunction=cF,budget=100, vectorized=TRUE)) ##target function is "vectorized", expects list of solutions as input
res
Description

Model-based optimization for combinatorial or mixed problems. Based on measures of distance or dissimilarity.

Usage

optimCEGO(x = NULL, fun, control = list())

Arguments

x  Optional initial design as a list. If NULL (default), creationFunction (in control list) is used to create initial design. If x has less individuals than specified by control$evalInit, creationFunction will fill up the design.

fun  target function to be minimized

control (list), with the options of optimization and model building approaches employed:

evalInit  Number of initial evaluations (i.e., size of the initial design), integer, default is 2

vectorized  Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE

verbosity  Level of text output during run. Defaults to 0, no output.

plotting  Plot optimization progress during run (TRUE) or not (FALSE). Default is FALSE.

targetY  optimal value to be found, stopping criterion, default is -Inf

budget  maximum number of target function evaluations, default is 100

creationRetries  When a model does not predict an actually improving solution, a random exploration step is performed. creationRetries solutions are created randomly. For each, distance to all known solutions is calculated. The minimum distance is recorded for each random solution. The random solution with maximal minimum distance is chosen to be evaluated in the next iteration.

model  Model to be used as a surrogate of the target function. Default is "K" (Kriging). Also available are: "LM" (linear, distance-based model), "RBFN" (Radial Basis Function Network).

modelSettings  List of settings for model building, passed on as the control argument to the model training functions modelKriging, modellinear, modelRBFN.

infill  This parameter specifies a function to be used for the infill criterion (e.g., the default is expected improvement infillExpectedImprovement). To use no specific infill criterion this has to be set to NA, in which case the prediction of the surrogate model is used. Infill criteria are only used with models that may provide some error estimate with predictions.
optimizer Optimizer that finds the minimum of the surrogate model. Default is optimEA, an Evolutionary Algorithm.

optimizerSettings List of settings (control) for the optimizer function.

initialDesign Design function that generates the initial design. Default is designMaxMinDist, which creates a design that maximizes the minimum distance between points.

initialDesignSettings List of settings (control) for the initialDesign function.

creationFunction Function to create individuals/solutions in search space. Default is a function that creates random permutations of length 6

distanceFunction distanceFunction a suitable distance function of type \( f(x_1,x_2) \), returning a scalar distance value, preferably between 0 and 1. Maximum distances larger 1 are not a problem, but may yield scaling bias when different measures are compared. Should be non-negative and symmetric. With the setting control$model="K" this can also be a list of different fitness functions. Default is Hamming distance for permutations: distancePermutationHamming.

Value

a list:

xbest best solution found

ybest fitness of the best solution

x history of all evaluated solutions

y corresponding target function values \( f(x) \)

fit model-fit created in the last iteration

fpred prediction function created in the last iteration

count number of performed target function evaluations

message message string, giving information on termination reason

convergence error/status code: -1 for termination due to failed model building, 0 for termination due to depleted budget, 1 if attained objective value is equal to or below target (control$targetY)

References


See Also

modelKriging, modelLinear, modelRBFN, buildModel, optimEA
Examples

```r
seed <- 0
# distance
dF <- distancePermutationHamming
# mutation
mF <- mutationPermutationSwap
# recombination
rF <- recombinationPermutationCycleCrossover
# creation
cF <- function() sample(5)
# objective function
lF <- landscapeGeneratorUNI(1:5, dF)
# start optimization
set.seed(seed)
res1 <- optimCEGO(lF, list(
    creationFunction = cF,
    distanceFunction = dF,
    optimizerSettings = list(budget = 100, popsize = 10,
                            mutationFunction = mF, recombinationFunction = rF),
    evalInit = 5, budget = 15, targetY = 0, verbosity = 1, model = modelKriging,
    vectorized = TRUE))  ## target function is "vectorized", expects list as input
set.seed(seed)
res2 <- optimCEGO(lF, list(
    creationFunction = cF,
    distanceFunction = dF,
    optimizerSettings = list(budget = 100, popsize = 10,
                            mutationFunction = mF, recombinationFunction = rF),
    evalInit = 5, budget = 15, targetY = 0, verbosity = 1, model = modelRBFN,
    vectorized = TRUE))  ## target function is "vectorized", expects list as input
res1$xbest
res2$xbest
```

---

**optimEA**

*Evolutionary Algorithm for Combinatorial Optimization*

**Description**

A basic implementation of a simple Evolutionary Algorithm for Combinatorial Optimization. Default evolutionary operators aim at permutation optimization problems.

**Usage**

```r
optimEA(x = NULL, fun, control = list())
```

**Arguments**

- `x`: Optional start individual(s) as a list. If NULL (default), `creationFunction` (in control list) is used to create initial design. If `x` has less individuals than the population size, `creationFunction` will fill up the rest.
fun

target function to be minimized

(list), with the options:

control

budget The limit on number of target function evaluations (stopping criterion)
(default: 1000).

popsize Population size (default: 100).

generations Number of generations (stopping criterion) (default: Inf).

targetY Target function value (stopping criterion) (default: -Inf).

vectorized Boolean. Defines whether target function is vectorized (takes a
list of solutions as argument) or not (takes single solution as argument).

Default: FALSE.

verbosity Level of text output during run. Defaults to 0, no output.

plotting Plot optimization progress during run (TRUE) or not (FALSE). De-
dfault is FALSE.

archive Whether to keep all candidate solutions and their fitness in an archive
(TRUE) or not (FALSE). Default is TRUE. New solutions that are identical
to an archived one, will not be evaluated. Instead, their fitness is taken from
the archive.

recombinationFunction Function that performs recombination, default: recombinationPermutationCycleCrossover,
which is cycle crossover for permutations.

recombinationRate Number of offspring, defined by the fraction of the popu-
lation (popsize) that will be recombined.

mutationFunction Function that performs mutation, default: mutationPermutationSwap,
which is swap mutation for permutations.

parameters Default parameter list for the algorithm, e.g., mutation rate, etc.

selection Survival selection process: "tournament" (default) or "truncation".

tournamentSize Tournament size (default: 2).

tournamentProbability Tournament probability (default: 0.9).

localSearchFunction If specified, this function is used for a local search step.

Default is NULL.

localSearchRate Specifies on what fraction of the population local search is
applied. Default is zero. Maximum is 1 (100 percent).

localSearchSettings List of settings passed to the local search function con-
trol parameter.

stoppingCriterionFunction Custom additional stopping criterion. Function
evaluated on the population, receiving all individuals (list) and their fitness
(vector). If the result is FALSE, the algorithm stops.

verbosity >0 for text output.

creationFunction Function to create individuals/solutions in search space.

Default is a function that creates random permutations of length 6.

selfAdaption An optional ParamHelpers object, that describes parameters of
the optimization (see parameters) which are subject to self-adaption. An
example is given in mutationSelfAdapt.

selfAdaptTau Positive numeric value, that controls the learning rate of numer-
cical/integer self-adaptive parameters.

selfAdaptP Value in $[0,1]$. A probability of mutation for all categorical, self-
adaptive parameters.
**Value**

a list:

xbest best solution found.

ybest fitness of the best solution.

x history of all evaluated solutions.

y corresponding target function values f(x).

count number of performed target function evaluations.

message Termination message: Which stopping criterion was reached.

population Last population.

fitness Fitness of last population.

**See Also**

optimCEGO, optimRS, optim2Opt, optimMaxMinDist

**Examples**

```r
#First example: permutation optimization
seed=0
#distance
dF <- distancePermutationHamming
#mutation
mF <- mutationPermutationSwap
#recombination
rF <- recombinationPermutationCycleCrossover
#creation
cF <- function()sample(5)
#objective function
lF <- landscapeGeneratorUNI(1:5,dF)
#start optimization
set.seed(seed)
res <- optimEA(lF,list(creationFunction=cF,mutationFunction=mF,recombinationFunction=rF,
popsize=6,budget=60,targetY=0,verbosity=1,vectorized=TRUE)) ##target function is "vectorized", expects list as input
res$xbest

#Second example: binary string optimization
#number of bits
N <- 50
#target function (simple example)
f <- function(x){
  sum(x)
}
#function to create random Individuals
cf <- function(){
  sample(c(FALSE,TRUE),N,replace=TRUE)
}
#control list
cntrl1 <- list(

```
optimInterface

Optimization Interface (continuous, bounded)

Description

This function is an interface fashioned like the `optim` function. Unlike `optim`, it collects a set of bound-constrained optimization algorithms with local as well as global approaches. It is, e.g., used in the CEGO package to solve the optimization problem that occurs during parameter estimation in the Kriging model (based on Maximum Likelihood Estimation). Note that this function is NOT applicable to combinatorial optimization problems.

Usage

```r
optimInterface(x, fun, lower = -Inf, upper = Inf, control = list(), ...)
```

Arguments

- **x** is a point (vector) in the decision space of `fun`
- **fun** is the target function of type `y = f(x,...)`
- **lower** is a vector that defines the lower boundary of search space
- **upper** is a vector that defines the upper boundary of search space
- **control** is a list of additional settings. See details.
- **...** additional parameters to be passed on to `fun`

Details

The control list contains:

- `funEvals` stopping criterion, number of evaluations allowed for `fun` (defaults to 100)
- `reltol` stopping criterion, relative tolerance (default: 1e-6)
factr stopping criterion, specifying relative tolerance parameter factr for the L-BFGS-B method in the optim function (default: 1e10)

popsiz e population size or number of particles (default: 10*dimension, where dimension is derived from the length of the vector lower).

restarts whether to perform restarts (Default: TRUE). Restarts will only be performed if some of the evaluation budget is left once the algorithm stopped due to some stopping criterion (e.g., retol).

method will be used to choose the optimization method from the following list: "L-BFGS-B" - BFGS quasi-Newton: stats Package optim function
"nlminb" - box-constrained optimization using PORT routines: stats Package nlminb function
"DEoptim" - Differential Evolution implementation: DEoptim Package
Additionally to the above methods, several methods from the package nloptr can be chosen. The complete list of suitable nlopt methods (non-gradient, bound constraints) is:
"NLOPT_GN_DIRECT","NLOPT_GN_DIRECT_L","NLOPT_GN_DIRECT_L_RAND","NLOPT_GN_DIRECT_M
"NLOPT_GN_ORIG_DIRECT","NLOPT_GN_ORIG_DIRECT_L","NLOPT_LN_PRAXIS",
"NLOPT_GN_CRS2_LM","NLOPT_LN_COBYLA","NLOPT_LN_NELDERMEAD","NLOPT_LN_SBPLX","NLOPT_LNLayoutConstraint

All of the above methods use bound constraints. For references and details on the specific methods, please check the documentation of the packages that provide them.

Value

This function returns a list with:

xbest parameters of the found solution
ybest target function value of the found solution
count number of evaluations of fun

optimMaxMinDist  Max-Min-Distance Optimizer

Description

One-shot optimizer: Create a design with maximum sum of distances, and evaluate. Best candidate is returned.

Usage

optimMaxMinDist(x = NULL, fun, control = list())
Arguments

- **x**: Optional set of solution(s) as a list, which are added to the randomly generated solutions and are also evaluated with the target function.
- **fun**: Target function to be minimized
- **control** (list), with the options:
  - **budget**: The limit on number of target function evaluations (stopping criterion) (default: 100).
  - **vectorized**: Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE.
  - **creationFunction**: Function to create individuals/solutions in search space. Default is a function that creates random permutations of length 6.
  - **designBudget**: Budget of the design function `designMaxMinDist`, which is the number of randomly created candidates in each iteration.

Value

A list:

- **xbest**: Best solution found
- **ybest**: Fitness of the best solution
- **x**: History of all evaluated solutions
- **y**: Corresponding target function values \( f(x) \)
- **count**: Number of performed target function evaluations

See Also

- `optimCEGO`, `optimEA`, `optimRS`, `optim2Opt`

Examples

```r
seed=0
# distance
dF <- distancePermutationHamming
# creation
cF <- function()sample(5)
# objective function
lF <- landscapeGeneratorUNI(1:5,dF)
# start optimization
set.seed(seed)
res <- optimMaxMinDist(lF,list(creationFunction=cF,budget=20, vectorized=TRUE)) ## target function is "vectorized", expects list as input
res$xbest
```
Description

An optimization algorithm from the family of Evolution Strategies, designed to optimize mixed-integer problems: The search space is composed of continuous (real-valued) parameters, ordinal integers and categorical parameters. Please note that the categorical parameters need to be coded as integers (type should not be a factor or character). It is an implementation (with a slight modification) of MIES as described by Li et al. (2013). Note, that this algorithm always has a step size for each solution parameter, unlike Li et al., we did not include the option to change to a single step-size for all parameters. Dominant recombination is used for solution parameters (the search space parameters), intermediate recombination for strategy parameters (i.e., step sizes). Mutation: Self-adaptive, step sizes sigma are optimized alongside the solution parameters. Real-valued parameters are subject to variation based on independent normal distributed random variables. Ordinal integers are subject to variation based on the difference of geometric distributions. Categorical parameters are changed at random, with a self-adapted probability. Note, that a more simple bound constraint method is used. Instead of the Transformation $T_{a,b}(x)$ described by Li et al., optimMIES simply replaces any value that exceeds the bounds by respective boundary value.

Usage

optimMIES(x = NULL, fun, control = list())

Arguments

x Optional start individual(s) as a list. If NULL (default), creationFunction (in control list) is used to create initial design. If x has less individuals than the population size, creationFunction will fill up the rest.

fun target function to be minimized.

control (list), with the options:

budget The limit on number of target function evaluations (stopping criterion) (default: 1000).
popsize Population size (default: 100).
generations Number of generations (stopping criterion) (default: Inf).
targetY Target function value (stopping criterion) (default: -Inf).
vectorized Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE.

verbosity Level of text output during run. Defaults to 0, no output.

plotting Plot optimization progress during run (TRUE) or not (FALSE). Default is FALSE.

archive Whether to keep all candidate solutions and their fitness in an archive (TRUE) or not (FALSE). Default is TRUE.
stoppingCriterionFunction
Custom additional stopping criterion. Function evaluated on the population, receiving all individuals (list) and their fitness (vector). If the result is FALSE, the algorithm stops.

types  A vector that specifies the data type of each variable: "numeric", "integer" or "factor".

lower  Lower bound of each variable. Factor variables can have the lower bound set to NA.

upper  Upper bound of each variable. Factor variables can have the upper bound set to NA.

levels List of levels for each variable (only relevant for categorical variables). Should be a vector of numerical values, usually integers, but not necessarily a sequence. HAS to be given if any factors/categoricals are present. Else, set to NA.

Details
The control variables types, lower, upper and levels are especially important.

Value
a list:
xbest  best solution found.
ybest  fitness of the best solution.
  x  history of all evaluated solutions.
y  corresponding target function values f(x).
count  number of performed target function evaluations.
message  Termination message: Which stopping criterion was reached.
population  Last population.
fitness  Fitness of last population.

References

See Also
optimCEGO, optimRS, optimEA, optim2Opt, optimMaxMinDist

Examples
set.seed(1)
controlList <- list(lower=c(-5,-5,1,1,NA,NA),upper=c(10,5,10,10,NA,NA),
types=c("numeric","numeric","integer","integer","factor","factor"),
levels=list(NA,NA,NA,NA,c(1,3,5),1:4),
vectorized = FALSE)
optimRS

Combinatorial Random Search

Description

Random Search for mixed or combinatorial optimization. Solutions are generated completely at random.

Usage

optimRS(x = NULL, fun, control = list())

Arguments

x Optional set of solution(s) as a list, which are added to the randomly generated solutions and are also evaluated with the target function.
fun target function to be minimized
control (list), with the options:
  budget The limit on number of target function evaluations (stopping criterion)
  (default: 100)
predict.modelKriging

vectorized Boolean. Defines whether target function is vectorized (takes a list of solutions as argument) or not (takes single solution as argument). Default: FALSE

creationFunction Function to create individuals/solutions in search space. Default is a function that creates random permutations of length 6

Value

a list:

xbest best solution found
ybest fitness of the best solution
x history of all evaluated solutions
y corresponding target function values f(x)
count number of performed target function evaluations

See Also

optimCEGO, optimEA, optim2Opt, optimMaxMinDist

Examples

seed=0
#distance
dF <- distancePermutationHamming
#creation
cF <- function()sample(5)
#objective function
lF <- landscapeGeneratorUNI(1:5,dF)
#start optimization
set.seed(seed)
res <- optimRS(,lF,list(creationFunction=cF,budget=100,
vectorized=TRUE)) ##target function is "vectorized", expects list as input
res$xbest

predict.modelKriging  Krigeing Prediction

Description

Predict with a model fit resulting from modelKriging.

Usage

## S3 method for class 'modelKriging'
predict(object, x, ...)
**Arguments**

- **object** - fit of the Kriging model (settings and parameters), of class `modelKriging`.
- **x** - list of samples to be predicted
- **...** - further arguments, not used

**Value**

Returned value depends on the setting of `object$predAll`

- **TRUE**: list with function value (mean) `object$y` and uncertainty estimate `object$s` (standard deviation)
- **FALSE**: `object$y` only

**See Also**

- `modelKriging`
- `simulate.modelKriging`

---

**predict.modelLinear**  
*Predict: Combinatorial Kriging*

**Description**

Predict with a `modelLinear` fit.

**Usage**

```r
## S3 method for class 'modelLinear'
predict(object, x, ...)
```

**Arguments**

- **object** - fit of the Kriging model (settings and parameters), of class `modelLinear`.
- **x** - list of samples to be predicted
- **...** - further arguments, not used

**Value**

numeric vector of predictions

**See Also**

- `modelLinear`
predict.modelRBFN  Predict: Combinatorial RBFN

Description

Predict with a model fit resulting from modelRBFN.

Usage

```r
## S3 method for class 'modelRBFN'
predict(object, x, ...)
```

Arguments

- `object` fit of the RBFN model (settings and parameters), of class modelRBFN.
- `x` list of samples to be predicted
- `...` further arguments, not used

Value

Returned value depends on the setting of `object$predAll`
- `TRUE`: list with function value (mean) $y$ and uncertainty estimate $s$ (standard deviation)
- `FALSE`: $y$ only

See Also

modelRBFN

recombinationBinary1Point  Single Point Crossover for Bit Strings

Description

Given a population of bit-strings, this function recombines each individual with another individual by randomly specifying a single position. Information before that position is taken from the first parent, the rest from the second.

Usage

```r
recombinationBinary1Point(population, parameters)
```

Arguments

- `population` List of bit-strings
- `parameters` not used
**recombinationBinary2Point**

*Two Point Crossover for Bit Strings*

**Description**
Given a population of bit-strings, this function recombines each individual with another individual by randomly specifying 2 positions. Information in-between is taken from one parent, the rest from the other.

**Usage**
```
recombinationBinary2Point(population, parameters)
```

**Arguments**
- **population**: List of bit-strings
- **parameters**: not used

**Value**
population of recombined offspring

---

**recombinationBinaryAnd**

*Arithmetic (AND) Crossover for Bit Strings*

**Description**
Given a population of bit-strings, this function recombines each individual with another individual by computing parent1 & parent2 (logical AND).

**Usage**
```
recombinationBinaryAnd(population, parameters)
```

**Arguments**
- **population**: List of bit-strings
- **parameters**: not used

**Value**
population of recombined offspring
**recombinationBinaryUniform**

*Uniform Crossover for Bit Strings*

**Description**

Given a population of bit-strings, this function recombines each individual with another individual by randomly picking bits from each parent. Note, that optimEA will not pass the whole population to recombination functions, but only the chosen parents.

**Usage**

recombinationBinaryUniform(population, parameters)

**Arguments**

- **population**: List of bit-strings
- **parameters**: not used

**Value**

population of recombined offspring

---

**recombinationPermutationAlternatingPosition**

*Alternating Position Crossover (AP) for Permutations*

**Description**

Given a population of permutations, this function recombines each individual with another individual. Note, that optimEA will not pass the whole population to recombination functions, but only the chosen parents.

**Usage**

recombinationPermutationAlternatingPosition(population, parameters)

**Arguments**

- **population**: List of permutations
- **parameters**: not used

**Value**

population of recombined offspring
recombinationPermutationCycleCrossover

*Cycle Crossover (CX) for Permutations*

**Description**

Given a population of permutations, this function recombines each individual with another individual. Note, that `optimEA` will not pass the whole population to recombination functions, but only the chosen parents.

**Usage**

```
recombinationPermutationCycleCrossover(population, parameters)
```

**Arguments**

- `population` | List of permutations
- `parameters` | not used

**Value**

- population of recombined offspring

---

recombinationPermutationOrderCrossover1

*Order Crossover 1 (OX1) for Permutations*

**Description**

Given a population of permutations, this function recombines each individual with another individual. Note, that `optimEA` will not pass the whole population to recombination functions, but only the chosen parents.

**Usage**

```
recombinationPermutationOrderCrossover1(population, parameters)
```

**Arguments**

- `population` | List of permutations
- `parameters` | not used

**Value**

- population of recombined offspring
recombinationPermutationPositionBased

Position Based Crossover (POS) for Permutations

Description

Given a population of permutations, this function recombines each individual with another individual. Note, that `optimEA` will not pass the whole population to recombination functions, but only the chosen parents.

Usage

`recombinationPermutationPositionBased(population, parameters)`

Arguments

- `population` List of permutations
- `parameters` not used

Value

population of recombined offspring

recombinationSelfAdapt

Self-adaptive recombination operator

Description

This recombination function selects an operator (provided in `parameters$recombinationFunctions`) based on self-adaptive parameters chosen for each individual separately.

Usage

`recombinationSelfAdapt(population, parameters)`

Arguments

- `population` List of permutations
- `parameters` list, contains the available single mutation functions (`mutationFunctions`), and a `data.frame` that collects the chosen function and mutation rate for each individual (`selfAdapt`).

See Also

`optimEA, mutationSelfAdapt`
**recombinationStringSinglePointCrossover**

*Single Point Crossover for Strings*

**Description**

Given a population of strings, this function recombines each individual with another random individual. Note, that `optimEA` will not pass the whole population to recombination functions, but only the chosen parents.

**Usage**

```
recombinationStringSinglePointCrossover(population, parameters)
```

**Arguments**

- **population**: List of strings
- **parameters**: not used

**Value**

population of recombined offspring

**repairConditionsCorrelationMatrix**

*Repair Conditions of a Correlation Matrix*

**Description**

This function repairs correlation matrices, so that the following two properties are ensured: The correlations values should be between -1 and 1, and the diagonal values should be one.

**Usage**

```
repairConditionsCorrelationMatrix(mat)
```

**Arguments**

- **mat**: symmetric, PSD distance matrix. If your matrix is not CNSD, use `correctionDefinite` first. Or use `correctionKernelMatrix`.

**Value**

repaired correlation matrix
repairConditionsDistanceMatrix

References


See Also

correctionDefinite, correctionDistanceMatrix, correctionKernelMatrix, correctionCNSD, repairConditionsDistanceMatrix

Examples

```r
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x,distancePermutationInsert)
K <- exp(-0.01*D)
K <- correctionDefinite(K,type="PSD")$mat
K <- repairConditionsCorrelationMatrix(K)
```

Repair Conditions of a Distance Matrix

Description

This function repairs distance matrices, so that the following two properties are ensured: The distance values should be non-zero and the diagonal should be zero. Other properties (conditionally negative semi-definite (CNSD), symmetric) are assumed to be given.

Usage

```r
repairConditionsDistanceMatrix(mat)
```

Arguments

- `mat` symmetric, CNSD distance matrix. If your matrix is not CNSD, use `correctionCNSD` first. Or use `correctionDistanceMatrix`.

Value

repaired distance matrix

References

simulate.modelKriging

Kriging Simulation

Description

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

Usage

```r
## S3 method for class 'modelKriging'
simulate(
  object, nsim = 1, seed = NA, xsim, conditionalSimulation = TRUE, returnAll = FALSE, ...
)
```

Arguments

- `object`: fit of the Kriging model (settings and parameters), of class `modelKriging`.
- `nsim`: number of simulations.
- `seed`: random number generator seed. Defaults to NA, in which case no seed is set.
- `xsim`: list of samples in input space, to be simulated.
- `conditionalSimulation`: logical, if set to TRUE (default), the simulation is conditioned with the training data of the Kriging model. Else, the simulation is non-conditional.
- `returnAll`: if set to TRUE, a list with the simulated values (y) and the corresponding covariance matrix (covar) of the simulated samples is returned.
- `...`: further arguments, not used.

Examples

```r
x <- list(c(2,1,4,3),c(2,4,3,1),c(4,2,1,3),c(4,3,2,1),c(1,4,3,2))
D <- distanceMatrix(x,distancePermutationInsert)
D <- correctionCNSD(D)
D <- repairConditionsDistanceMatrix(D)
```

See Also

- `correctionDefinite`, `correctionDistanceMatrix`, `correctionKernelMatrix`, `correctionCNSD`, `repairConditionsCorrelationMatrix`
solutionFunctionGeneratorBinary

Binary String Generator Function

Description

Returns a function that generates random bit-strings of length N. Can be used to create individuals of NK-Landscapes or other problems with binary representation.

Usage

solutionFunctionGeneratorBinary(N)

Arguments

N length of the bit-strings

Value

returns a function, without any arguments

References


See Also

modelKriging, predict.modelKriging
**solutionFunctionGeneratorPermutation**

*Permutation Generator Function*

**Description**

Returns a function that generates random permutations of length N. Can be used to generate individual solutions for permutation problems, e.g., Travelling Salesperson Problem.

**Usage**

```r
solutionFunctionGeneratorPermutation(N)
```

**Arguments**

- `N`: length of the permutations returned

**Value**

returns a function, without any arguments

**Examples**

```r
fun <- solutionFunctionGeneratorPermutation(10)
fun()
fun()
fun()
```

---

**solutionFunctionGeneratorString**

*String Generator Function*

**Description**

Returns a function that generates random strings of length N, with given letters. Can be used to generate individual solutions for permutation problems, e.g., Travelling Salesperson Problem.

**Usage**

```r
solutionFunctionGeneratorString(N, lts = c("A", "C", "G", "T"))
```

**Arguments**

- `N`: length of the permutations returned
- `lts`: letters allowed in the string
Value

returns a function, without any arguments

Examples

```r
fun <- solutionFunctionGeneratorString(10,c("A","C","G","T"))
fun()
fun()
fun()
```
controlModel (list), with the options for the model building procedure, it will be passed to the modelKriging function.

controlSimulation (list), with the parameters of the simulation:
  nsim the number of simulations, or test functions, to be created.
  conditionalSimulation whether (TRUE) or not (FALSE) to use conditional simulation.
  simulationSeed a random number generator seed. Defaults to NA; which means no seed is set. For sake of reproducibility, set this to some integer value.

Value

a list with the following elements: fun is a list of functions, where each function is the interpolation of one simulation realization. The length of the list depends on the nsim parameter. fit is the result of the modeling procedure, that is, the model fit of class modelKriging.

References


See Also

modelKriging, simulate.modelKriging, createSimulatedTestFunction.

Examples

```r
nsim <- 10
seed <- 12345
n <- 6
set.seed(seed)
# target function:
fun <- function(x){
  exp(-20*x) + sin(6*x^2) + x
}
# "vectorize" target
f <- function(x){sapply(x,fun)}
dF <- function(x,y)(sum((x-y)^2)) # sum of squares
# plot params
par(mfrow=c(4,1),mar=c(2.3,2.5,0.2,0.2),mgp=c(1.4,0.5,0))
# test samples for plots
xtest <- as.list(seq(from=-0,by=0.005,to=1))
plot(xtest,f(xtest),type="l",xlab="x",ylab="Obj. function")
# evaluation samples (training)
```
xb <- as.list(runif(n))
yb <- f(xb)
# support samples for simulation
x <- as.list(sort(c(runif(100),unlist(xb)))))
# fit the model and simulate:
res <- testFunctionGeneratorSim(xb,yb,x,dF,
  list(algThetaControl=list(method="NLOPT_GN_DIRECT_L",funEvals=100),
       useLambda=FALSE),
  list(nsim=nsim,conditionalSimulation=FALSE))
fit <- res$fit
fun <- res$fun
# predicted obj. function values
ypred <- predict(fit,as.list(xtest))$y
plot(unlist(xtest),ypred,type="l",xlab="x",ylab="Estimation")
points(unlist(xb),yb,pch=19)

# plot non-conditional simulation

ynew <- NULL
for(i in 1:nsim)
ynew <- cbind(ynew,fun[[i]](xtest))
rangeY <- range(ynew)
plot(unlist(xtest),ynew[,1],type="l",ylim=rangeY,xlab="x",ylab="Simulation")
for(i in 2:nsim){
  lines(unlist(xtest),ynew[,i],col=i,type="l")
}

# create and plot test function, conditional

fun <- testFunctionGeneratorSim(xb,yb,x,dF,
  list(algThetaControl=
       list(method="NLOPT_GN_DIRECT_L",funEvals=100),
       useLambda=FALSE),
  list(nsim=nsim,conditionalSimulation=TRUE))$fun
ynew <- NULL
for(i in 1:nsim)
ynew <- cbind(ynew,fun[[i]](xtest))
rangeY <- range(ynew)
plot(unlist(xtest),ynew[,1],type="l",ylim=rangeY,xlab="x",ylab="Conditional sim.")
for(i in 2:nsim){
  lines(unlist(xtest),ynew[,i],col=i,type="l")
}
points(unlist(xb),yb,pch=19)
Index

*Topic package
  CEGO-package, 3

benchmarkGeneratorFSP, 4, 7–9
benchmarkGeneratorMaxCut, 5
benchmarkGeneratorNKL, 6
benchmarkGeneratorQAP, 5, 7, 8, 9
benchmarkGeneratorTSP, 5, 7, 8, 9
benchmarkGeneratorWT, 5, 7, 8, 9
buildModel, 62

CEGO (CEGO-package), 3
CEGO-package, 3
correctionCNSD, 10, 12, 13, 59, 80, 81
correctionDefinite, 11, 12–14, 79–81
correctionDistanceMatrix, 12, 59, 80, 81
correctionKernelMatrix, 13, 79–81
createSimulatedTestFunction, 14, 85
designMaxMinDist, 68
distanceMatrix, 16
distanceNumericHamming, 17
distanceNumericLCStr, 18
distanceNumericLevenshtein, 18
distancePermutationAdjacency, 19
distancePermutationChebyshev, 20
distancePermutationCos, 21
distancePermutationEuclidean, 22
distancePermutationHamming, 23
distancePermutationInsert, 23
distancePermutationInterchange, 24
distancePermutationLCStr, 25
distancePermutationLee, 26
distancePermutationLevenshtein, 27
distancePermutationLex, 28, 45
distancePermutationManhattan, 29
distancePermutationPosition, 30
distancePermutationPosition2, 31
distancePermutationR, 32
distancePermutationSwap, 33
distanceRealEuclidean, 34
distanceStringHamming, 34
distanceStringLCStr, 35
distanceStringLevenshtein, 36
distanceVector, 36
infillExpectedImprovement, 37
is.CNSD, 37, 39, 40
is.NSD, 38, 39, 40
is.PSD, 38, 39, 40
kernelMatrix, 41
landscapeGeneratorGaussian, 41, 43, 44
landscapeGeneratorMUL, 43, 44
landscapeGeneratorUNI, 43, 44
lexicographicPermutationOrderNumber, 28, 45
modelKriging, 10, 11, 14, 15, 46, 61, 62, 72, 73, 81, 82, 85
modellinear, 49, 61, 62, 73
modelRBFN, 50, 61, 62, 74
mutationBinaryBitFlip, 52
mutationBinaryBlockInversion, 53
mutationBinaryCycle, 53
mutationBinarySingleBitFlip, 54
mutationPermutationInsert, 54
mutationPermutationInterchange, 55
mutationPermutationReversal, 55
mutationPermutationSwap, 56, 64
mutationSelfAdapt, 56, 64, 78
mutationStringRandomChange, 57

nearCNSD, 58
nearPD, 58, 59
norm, 58

optim, 66
optim2Opt, 59, 65, 68, 70, 72
optimCEGO, 4, 60, 61, 65, 68, 70, 72

87
optimEA, 57, 60, 62, 63, 68, 70, 72, 76–79
optimInterface, 46, 66
optimMaxMinDist, 60, 65, 67, 70, 72
optimMIES, 69
optimRS, 60, 65, 68, 70, 71

predict.modelKriging, 48, 72, 82
predict.modelLinear, 50, 73
predict.modelRBFN, 52, 74

recombinationBinary1Point, 74
recombinationBinary2Point, 75
recombinationBinaryAnd, 75
recombinationBinaryUniform, 76
recombinationPermutationAlternatingPosition, 76
recombinationPermutationCycleCrossover, 64, 77
recombinationPermutationOrderCrossover1, 77
recombinationPermutationPositionBased, 78
recombinationSelfAdapt, 57, 78
recombinationStringSinglePointCrossover, 79

repairConditionsCorrelationMatrix, 13, 14, 79, 81
repairConditionsDistanceMatrix, 12, 13, 80, 80

simulate.modelKriging, 15, 73, 81, 85
solutionFunctionGeneratorBinary, 82
solutionFunctionGeneratorPermutation, 83
solutionFunctionGeneratorString, 83

testFunctionGeneratorSim, 14, 15, 84