Package ‘CEGO’

October 6, 2015

Version 2.0.0
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
Suggests nloptr
ByteCompile TRUE
Date 2015-10-06
Date/Publication 2015-10-06 15:56:36
NeedsCompilation yes
Repository CRAN

R topics documented:

CEGO-package .................................................. 3
benchmarkGeneratorFSP .................................... 4
benchmarkGeneratorNKL .................................... 5
benchmarkGeneratorQAP .................................... 6
benchmarkGeneratorTSP .................................... 7
benchmarkGeneratorWT .................................... 8
distanceMatrix ............................................. 9
distancePermutationAdjacency .......................... 9
distancePermutationChebyshev .......................... 10
distancePermutationCos ................................. 11
R topics documented:

distancePermutationEuclidean ............................... 12
distancePermutationHamming .................................. 13
distancePermutationInsert .................................... 13
distancePermutationInterchange ............................ 14
distancePermutationLCStr .................................... 15
distancePermutationLee ...................................... 16
distancePermutationLevenshtein ............................. 17
distancePermutationLex ...................................... 18
distancePermutationManhattan .............................. 19
distancePermutationPosition ................................. 20
distancePermutationPosition2 ............................... 21
distancePermutationR ......................................... 22
distancePermutationSwap ..................................... 23
distanceVector .................................................. 24
infillExpectedImprovement ................................. 24
landscapeGeneratorGaussian .................................. 25
landscapeGeneratorMUL ....................................... 26
landscapeGeneratorUNI ....................................... 27
lexicographicPermutationOrderNumber ....................... 28
modelKriging ................................................... 29
modelLinear .................................................... 32
modelRBFN ..................................................... 33
mutationBinary ............................................... 35
mutationBinaryFast ......................................... 35
mutationPermutationInterchange ............................ 36
mutationPermutationReversal ................................ 36
mutationPermutationSwap .................................... 37
optim2Opt ..................................................... 37
optimCEGO ..................................................... 39
optimEA ......................................................... 41
optimInterface ............................................... 43
optimRS ........................................................ 44
predict.modelKriging ........................................ 46
predict.modelLinear ......................................... 46
predict.modelRBFN .......................................... 47
recombinationBinaryUniformCrossoverFast .................... 48
recombinationPermutationCycleCrossover ..................... 48
solutionFunctionGeneratorBinary ............................ 49
solutionFunctionGeneratorPermutation ....................... 49

Index 50
Combinatorial Efficient Global Optimization in R

Description

Combinatorial Efficient Global Optimization

Details

Model building, surrogate model based optimization and Efficient Global Optimization in combinatorial or mixed search spaces.

Package: CEGO
Type: Package
Version: 2.0.0
Date: 2015-10-06
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 <mzaefferer@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.

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 cost=f(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)
**benchmarkGeneratorNKL**  
*NK-Landscape Benchmark Creation*

**Description**

Function that generates a NK-Landscapes.

**Usage**

```r
test <- benchmarkGeneratorNKL(N = 10, K = 1, PI = 1:K, g = NULL)
```

**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 <- benchmarkGeneratorNKL(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 <- benchmarkGeneratorNKL(6,3)
fun(c(1,0,1,1,0,0))
fun(c(0,1,1,1,0,0))
fun(c(1,0,1,1,1,0))
fun(c(1,0,1,1,1,1))
fun(c(1,0,1,1,0,0), g=matrix(runif(48),6))
fun(sample(c(0,1),6,TRUE))
```
benchmarkGeneratorQAP  \textit{Create Quadratic Assignment Problem (QAP) Benchmark}

**Description**

Creates a benchmark function for the Quadratic Assignment Problem.

**Usage**

\[ \text{benchmarkGeneratorQAP}(a, b) \]

**Arguments**

- \(a\) distance matrix
- \(b\) flow matrix

**Value**

the function of type cost=f(permutation)

**See Also**

\texttt{benchmarkGeneratorFSP}, \texttt{benchmarkGeneratorTSP}, \texttt{benchmarkGeneratorWT}

**Examples**

```
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)
    }
  }
}
# create a distance matrix
locations <- matrix(runif(n*2)*10,2)
B <- as.matrix(dist(locations))
# create QAP objective function
fun <- benchmarkGeneratorQAP(A,B)
# evaluate
fun(1:n)
fun(n: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**

\[
\text{benchmarkGeneratorTSP(distanceMatrix, type = "Cycle")}
\]

**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 \( \text{cost=} f(\text{permutation}) \)

**See Also**

`benchmarkGeneratorQAP, benchmarkGeneratorFSP, benchmarkGeneratorWT`

**Examples**

```r
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(1:n)
fun2(1:n)
```
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)
#weights
w <- sample(10,n,replace=TRUE)
#due dates
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)
**distanceMatrix**

*Calculate Distance Matrix*

**Description**

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

**Usage**

\[
\text{distanceMatrix}(x, \text{distFun})
\]

**Arguments**

- \(x\): list of samples, where each list element is a suitable input for \text{distFun}
- \(\text{distFun}\): Distance function of type \(f(x,y)=r\), where \(r\) is a scalar and \(x\) and \(y\) are elements whose distance is evaluated.

**Value**

The distance matrix

**Examples**

\[
\begin{align*}
x & \leftarrow \text{list}(5:1, c(2, 4, 5, 1, 3), c(5, 4, 3, 1, 2), \text{sample}(5)) \\
\text{distanceMatrix}(x, \text{distancePermutationHamming})
\end{align*}
\]

**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**

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

**Arguments**

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

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 <- 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

`distancePermutationChebyshev(x, y)`

Arguments

- `x`: first permutation (integer vector)
- `y`: second permutation (integer vector)
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
```r
distancePermutationCos(x, y)
```

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

Value
numeric distance value

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

References
Euclidean Distance for Permutations

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 \( r = \sqrt{\frac{2 \times 4 \times n \times (n + 1) \times (2 \times n + 1)}{6}} \) (if \( n \) is odd) or \( r = \sqrt{\frac{2 \times n \times (2 \times n - 1) \times (2 \times n + 1)}{3}} \) (if \( n \) is even).

Usage

distancePermutationEuclidean(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

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

\[
d_{Hamming}(x, y) = \frac{d(x, y)}{\max(d(x, y))}
\]

**Arguments**

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

**Value**

numeric distance value

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

**Examples**

```r
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. This distance is also called Ulam’s metric and can as well be interpreted to be based on the longest common subsequence of two permutations.
distancePermutationInterchange

Usage

distancePermutationInterchange(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)
distancePermutationInterchange(x, y)
p <- replicate(10, sample(1:5), simplify=FALSE)
distancematrix(p, distancePermutationInterchange)
```

---

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

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

**Arguments**

\[x\]  
first permutation (integer vector)

\[y\]  
second permutation (integer vector)

**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)

#
distancePermutationLee

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

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)
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

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)
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**

```r
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)
```
Manhattan Distance for Permutations

Description

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

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

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 \(r = \frac{(n^2)/2}{2}\) (if \(n\) is even).

Usage

\texttt{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

\begin{verbatim}
x <- 1:5
y <- c(5,1,2,3,4)
distancePermutationManhattan(x,y) p <- replicate(10,sample(1:5),simplify=FALSE) distanceMatrix(p,distancePermutationManhattan)
\end{verbatim}
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

A 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**

```
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(1,3,5,4,2)
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**

```r
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

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

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

**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)
```

**infillExpectedImprovement**

*Negative Logarithm of Expected Improvement*

**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**

\[
infillExpectedImprovement(mean, sd, min)
\]

**Arguments**

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

Create Gaussian Landscape

**Value**

Returns the negative logarithm of the Expected Improvement.

---

**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.

**Usage**

```r
gaussianGaussian(nGaussian = 10, upper = 1, ratio = 0.2, seed = 1, distanceFunction, creationFunction)
```

**Arguments**

- `nGaussian`: number of Gaussian components in the landscape. Default is 10.
- `upper`: upper boundary of the `distanceFunction`. Controls width of Gaussian components. Default is 1.
- `ratio`: minimal function value of the local minima. Default is 0.2. (Note: Global minimum will be at zero, local minimal will be in range \([\text{ratio}, 1]\)).
- `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**

Examples

```r
# 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
set.seed(seed)
#landscape
lf <- landscapeGeneratorUNI(cF(),df)
x <- seq(from=-0,by=0.001,to=1)
plot(x,lf(x),type="l")
## multi modal distance landscape
# set seed
set.seed(seed)
#landscape
lf <- landscapeGeneratorMUL(replicate(5,cF(),FALSE),df)
x <- seq(from=-0,by=0.001,to=1)
plot(x,lf(x),type="l")
## glg landscape
#landscape
lf <- landscapeGeneratorGaussian(nGaussian=20,upper=1,
ratio=0.8,seed=seed,df,cF)
x <- seq(from=-0,by=0.001,to=1)
plot(x,lf(x),type="l")
```

---

**Description**

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

**Usage**

```r
landscapeGeneratorMUL(ref = list(1:10), distanceFunction)
```

**Arguments**

- `ref` list of reference individuals
- `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

```r
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. Hence, the reference individual is the optimum of the landscape.

Usage

```r
landscapeGeneratorUNI(ref = 1:10, distanceFunction)
```

Arguments

- `ref`  
  Reference individual
- `distanceFunction`  
  Distance function, used to evaluate $d(x,\text{ref})$, 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.
References


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)
```

---

**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

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)
lexicographicPermutationOrderNumber(1:7)
lexicographicPermutationOrderNumber(7:1)

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.
- 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.
  - optimizeP: boolean that specifies whether the exponents ($p$) should be optimized. Else they will be set to two.
  - useLambda: whether or not to use the regularization constant lambda (nugget effect). Default is FALSE.
  - lambdaLower: lower boundary for lambda, default is $-6$
modelKriging

lambda upper boundary for lambda, default is 0

distances 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.

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).

Value

an object of class modelKriging containing the options and determined parameters for the model:

x (see parameters)
y (see parameters)
lower (see parameters)
upper (see parameters)
algTheta (see parameters)
algThetaControl (see parameters)
optimizeP (see parameters)

theta activity or width parameter theta, a parameter of the correlation function determined with
MLE

log10Theta log10 theta (i.e. log10(theta))

lambda regularization constant (nugget) lambda

log10Lambda log10 of regularization constant (nugget) lambda (i.e. log10(lambda))

p exponent p, parameter of the correlation function determined with MLE (if optimizeP is TRUE)

ymu vector of observations y, minus MLE of mu

SSQ Maximum Likelihood Estimate (MLE) of model parameter sigma^2

mu MLE of model parameter mu

Psi correlation matrix Psi

Psinv inverse of Psi

nevals number of Likelihood evaluations during MLE of theta/lambda/p

scaling Default is FALSE. If TRUE: Distances values are divided by maximum distance to inde-
dependent of the scale of the distance function.

distanceFunctionIndexMLE If a list of several distance measures (distanceFunction) was given,
this parameter contains the index value of the measure chosen with MLE.

References

Forrester, Alexander I.J.; Sobester, Andras; Keane, Andy J. (2008). Engineering Design via Surro-

Zaefferer, Martin; Stork, Joerg; Friese, Martina; Fischbach, Andreas; Naujoks, Boris; Bartz-Beielstein,
2014 conference on Genetic and evolutionary computation (GECCO '14). ACM, New York, NY,
USA, 871-878. DOI=10.1145/2576768.2598282 http://doi.acm.org/10.1145/2576768.2598282

Zaefferer, Martin; Stork, Joerg; Bartz-Beielstein, Thomas. (2014). Distance Measures for Permu-
tations in Combinatorial Efficient Global Optimization. In Parallel Problem Solving from Nature -
PPSN XIII (p. 373-383). Springer International Publishing.
See Also

`predict.modelKriging`

Examples

```r
# Set random number generator seed
data.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,
                     control=list(algThetaControl=list(method="L-BFGS-B"),useLambda=FALSE))
# Predicted obj. function values
ypred <- predict(fit,xtest)$y
# Uncertainty estimate
fit$predall <- TRUE
spred <- predict(fit,xtest)$s
# Plot
plot(ytest,ypred,xlab="true value",ylab="predicted value",
     pch=20,xlim=c(0.3,1),ylim=c(min(ypred)-0.1,max(ypred)+0.1))
segments(ytest,ypred-spred,ytest,ypred+spred)
epsilon = 0.02
segments(ytest-epsilon,ypred-spred,ytest+epsilon,ypred-spred)
segments(ytest-epsilon,ypred+spred,ytest+epsilon,ypred+spred)
abline(0,1,lty=2)
# Use a different/custom optimizer (here: SANN) for maximum likelihood estimation:
# (Note: Bound constraints are recommended, to avoid Inf values.
# This is really just a demonstration. SANN does not respect bound constraints.)
optimizer1 <- function(x,fun,lower=NULL,upper=NULL,control=NULL,...){
  res <- optim(x,fun,method="SANN",control=list(maxit=100),...)
  list(xbest=res$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
modelRBFN

# Examples

```
# 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[-c(1:15)]
x <- 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 \( f(x_1, x_2) \), 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: \( \exp(-\beta \cdot D) \), where \( D \) is the distance matrix. If beta is not specified, the heuristic in fbeta will be used to determine it, which is default behavior.
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 building, 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


See Also

predict.modelRBFN

Examples

```r
#set random number generator seed
set.seed()
#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",
```
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**

```r
mutationBinary(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

---

Bit-flip Mutation for Bit-strings (Fast)

**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**

```r
mutationBinaryFast(population, parameters)
```

**Arguments**

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

**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**

```
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

---

**optim20pt**

*Two-Opt*

**Description**

Implementation of a Two-Opt local search.

**Usage**

```
optim20pt(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
    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

Value

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

References


Examples

```r
seed=0
glgseed=1
#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 <- optim2Opt(lf,list(creationFunction=cF,budget=100,
    vectorized=TRUE)) ##target function is "vectorized", expects list of solutions as input
res
```
optimCEGO Combinatorial Efficient Global Optimization

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), create design (in control list) is used to create initial design. If x has less individuals than specified by control$evalInit, create function will fill up the design.

fun target function to be minimized

c 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

evalBudget 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. 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 "EA" an Evolutionary Algorithm. No alternatives implemented yet.
optimizerSettings: List of settings for the method to optimize the model.

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

distanceFunction: Function to create a suitable distance function of type f(x1,x2), 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$mode"l="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
glgseed=1
#creation
dF <- distancePermutationHamming
#mutation
mF <- mutationPermutationSwap
#recombination
rF <- recombinationPermutationCycleCrossover
#creation
cF <- function()sample(5)
```
Description

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

Usage

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

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). De-
default is FALSE.
archive Whether to keep all candidate solutions and their fitness in an archive
(TRUE) or not (FALSE). Default is TRUE.
recombinationFunction Function that performs recombination, default: recombinationPermutationCycleCrossover,
which is cycle crossover for permutations.
recombinationParameters Parameter list for recombination (e.g., recombina-
tionParameters$recombinationRate => recombination rate, defaults to 0.5). List
is passed to recombinationFunction.
mutationFunction Function that performs mutation, default: mutationPermutationSwap,
which is swap mutation for permutations.
mutationParameters Parameter list for mutation (e.g., mutationParameters$mutationRate
=> mutation rate). List is passed to mutationFunction. Default: empty list.
selection 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 control parameter.
stoppingCriterionFunction Custom additional stopping criterion. Function
evaluated on the population, receiving all individuals (list) and their fitness (vec-
tor). 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
duplicateFunction Function that evaluates a list of solutions for duplicates.
Default is the duplicated function.
duplicateRemoval If set to "all" and archiving is on, new individuals are com-
pared against the complete archive to avoid duplicates. With "population", this
is limited to the current population.

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, optimRopt
Examples

```r
seed=0
glglseed=1
#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=15, budget=100, targetY=0, verbosity=1, 
vectorized=TRUE)) ##target function is "vectorized", expects list as input
res$x
```

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 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)
- `popsize` population size or number of particles (default: $10 \times \text{dimension}$, where dimension is derived from the length of the vector `lower`).
- `restarts` the number of restarts to perform (Default: 0). Function evaluation budget will be split accordingly. Early convergence of optimization runs may lead to additional restarts. Violations of the provided budget may decrease the number of restarts.

`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 `nloptr` methods (non-gradient, bound constraints) is:

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`

---

### optimRS

Combination of 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

c 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

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

Examples

seed=0
glgseed=1
#distance
dF <- distancePermutationHamming
#mutation
mF <- mutationPermutationSwap
#recombination
rF <- recombinationPermutationCycleCrossover
#create

lf <- function() sample(5)
#objective function

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  

Predict: Combinatorial Kriging

Description

Predict with a model fit resulting from `modelKriging`.

Usage

```r
## 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) $y$ and uncertainty estimate $s$ (standard deviation)
- `FALSE`: $y$ only

See Also

`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
predict.modelRBFN

Value

numeric vector of predictions

See Also

modelLinear

---

**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`
**RecombinationBinaryUniformCrossoverFast**

*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**

\[
\text{recombinationBinaryUniformCrossoverFast(population, parameters)}
\]

**Arguments**

- **population**: List of bit-strings
- **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 random individual. Note, that optimEA will not pass the whole population to recombination functions, but only the chosen parents.

**Usage**

\[
\text{recombinationPermutationCycleCrossover(population, parameters)}
\]

**Arguments**

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

**Value**

population of recombined offspring
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**

```rust
solutionFunctionGeneratorBinary(N)
```

**Arguments**

- **N**: length of the bit-strings

**Value**

returns a function, without any arguments

---

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**

```rust
solutionFunctionGeneratorPermutation(N)
```

**Arguments**

- **N**: length of the permutations returned

**Value**

returns a function, without any arguments
Index

*Topic package
  CEGO-package, 3

benchmarkGeneratorFSP, 4, 6–8
benchmarkGeneratorKWL, 5
benchmarkGeneratorQAP, 4, 6, 7, 8
benchmarkGeneratorTSP, 4, 6, 7, 8
benchmarkGeneratorWT, 4, 6, 7, 8
buildModel, 40

CEGO (CEGO-package), 3
CEGO-package, 3

distanceMatrix, 9
distancePermutationAdjacency, 9
distancePermutationChebyshev, 10
distancePermutationCos, 11
distancePermutationEuclidean, 12
distancePermutationHamming, 13
distancePermutationInsert, 13
distancePermutationInterchange, 14
distancePermutationLCStr, 15
distancePermutationLee, 16
distancePermutationLevenshtein, 17
distancePermutationLex, 18, 28
distancePermutationManhattan, 19
distancePermutationPosition, 20
distancePermutationPosition2, 21
distancePermutationR, 22
distancePermutationSwap, 23
distanceVector, 24

infillExpectedImprovement, 24

landscapeGeneratorGaussian, 25, 27, 28
landscapeGeneratorMUL, 26, 28
landscapeGeneratorUNI, 27, 27
lexicographicPermutationOrderNumber, 18, 28

modelKriging, 29, 39, 40, 46

modelLinear, 32, 39, 40, 47
modelRBFN, 33, 39, 40, 47
mutationBinary, 35
mutationBinaryFast, 35
mutationPermutationInterchange, 36
mutationPermutationReversal, 36
mutationPermutationSwap, 37, 42

optim, 43
optim2Opt, 37, 42
optimCEGO, 3, 39, 42, 45
optimEA, 40, 41, 45, 48
optimInterface, 29, 43
optimRS, 42, 44

predict.modelKriging, 31, 46
predict.modelLinear, 32, 46
predict.modelRBFN, 33, 47

recombinationBinaryUniformCrossoverFast, 48
recombinationPermutationCycleCrossover, 42, 48

solutionFunctionGeneratorBinary, 49
solutionFunctionGeneratorPermutation, 49