Package 'CEGO'

January 27, 2024

Version 2.4.3

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, anticlust

Suggests nloptr

ByteCompile TRUE

Date 2024-01-27

RoxygenNote 7.3.1

Encoding UTF-8

NeedsCompilation yes

Repository CRAN

Date/Publication 2024-01-27 21:30:03 UTC

R topics documented:

`CEGO-package` *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. This includes methods for distance calculation, modeling and handling of indefinite kernels/distances.

Package:	CEGO
Type:	Package
Version:	2.4.3
Date:	2024-01-27
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

Zaefferer, Martin; Stork, Joerg; Friese, Martina; Fischbach, Andreas; Naujoks, Boris; Bartz-Beielstein, Thomas. (2014). Efficient global optimization for combinatorial problems. In Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO '14). ACM, New York, NY, USA, 871-878. DOI=10.1145/2576768.2598282

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

Zaefferer, Martin and Bartz-Beielstein, Thomas (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

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
#ceate a matrix of processing times
A <- matrix(sample(n*m,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.

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

```
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 <- benchmarkGeneratorNKL(6,2,c(-1,1))
fun(c(1,0,1,1,0,0))
fun <- benchmarkGeneratorNKL(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))
```

---

benchmarkGeneratorQAP *Create Quadratic Assignment Problem (QAP) Benchmark*

Description

Creates a benchmark function for the Quadratic Assignment Problem.

Usage

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

Arguments

a	distance matrix
b	flow matrix

Value

the function of type cost=f(permutation)

See Also

benchmarkGeneratorFSP, benchmarkGeneratorTSP, benchmarkGeneratorWT

Examples

```
set.seed(1)
n=5
#ceate 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)
A[j,i] <- A[i,j]
```

```
  }
}
}
#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

```
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)
#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)
```

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

Value

the corrected CNSD matrix

References

Martin Zaefferer and Thomas Bartz-Beielstein. (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

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`	torelance 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

Martin Zaefferer and Thomas Bartz-Beielstein. (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

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.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

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

Value

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

References

Martin Zaefferer and Thomas Bartz-Beielstein. (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

See Also

`correctionDefinite`,`correctionCNSD`,`repairConditionsDistanceMatrix`

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 <- correctionDistanceMatrix(D)$mat
is.CNSD(D) #matrix should now be CNSD
D
```

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

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

Martin Zaefferer and Thomas Bartz-Beielstein. (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

See Also

`correctionDefinite`, `repairConditionsCorrelationMatrix`

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)
K <- exp(-0.01*D)
is.PSD(K) #matrix should not be PSD
K <- correctionKernelMatrix(K)$mat
is.PSD(K) #matrix should now be CNSD
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

```
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

`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

N. A. Cressie. Statistics for Spatial Data. JOHN WILEY & SONS INC, 1993.

C. Lantuejoul. Geostatistical Simulation - Models and Algorithms. Springer-Verlag Berlin Heidelberg, 2002.

Zaefferer, M.; Fischbach, A.; Naujoks, B. & Bartz-Beielstein, T. Simulation Based Test Functions for Optimization Algorithms Proceedings of the Genetic and Evolutionary Computation Conference 2017, ACM, 2017, 8.

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(
```

```
   algThetaControl=list(method="NLOPT_GN_DIRECT_L",funEvals=100),useLambda=FALSE))
fit
#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)
##############################
# create test function non conditional
##############################
fun <- createSimulatedTestFunction(x,fit,nsim,FALSE,seed=1)
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 test function conditional
##############################
fun <- createSimulatedTestFunction(x,fit,nsim,TRUE,seed=1)
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)
dev.off()
```

`distanceMatrix` *Calculate Distance Matrix*

Description

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

Usage

```
distanceMatrix(X, distFun, ...)
```

Arguments

`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

Value

The distance matrix

Examples

```
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

```
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
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

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)
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)

References

Sevaux, Marc, and Kenneth Soerensen. "Permutation distance measures for memetic algorithms with population management." Proceedings of 6th Metaheuristics International Conference (MIC'05). 2005.

Reeves, Colin R. "Landscapes, operators and heuristic search." Annals of Operations Research 86 (1999): 473-490.

Examples

```
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)

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

---

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

Singhal, Amit (2001)."Modern Information Retrieval: A Brief Overview". Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 24 (4): 35-43

Examples

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

`distancePermutationEuclidean` *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 $r = sqrt(2 * 4 * c * (c+1) * (2 * c+1)/6)$ with $c = (n-1)/2$ (if n is odd) or $r = sqrt(2 * c * (2 * c-1) * (2 * c+1)/3)$ with $c = n/2$ (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

```
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

Schiavinotto, Tommaso, and Thomas Stuetzle. "A review of metrics on permutations for search landscape analysis." Computers & operations research 34.10 (2007): 3143-3153.

Wikipedia contributors, "Longest increasing subsequence", Wikipedia, The Free Encyclopedia, 12 November 2014, 19:38 UTC, [accessed 13 November 2014]

Examples

```
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)
```

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

Schiavinotto, Tommaso, and Thomas Stuetzle. "A review of metrics on permutations for search landscape analysis." Computers & operations research 34.10 (2007): 3143-3153.

Examples

```
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

```
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)

References

Hirschberg, Daniel S. "A linear space algorithm for computing maximal common subsequences." Communications of the ACM 18.6 (1975): 341-343.

Examples

```
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)

References

Lee, C., "Some properties of nonbinary error-correcting codes," Information Theory, IRE Transactions on, vol.4, no.2, pp.77,82, June 1958

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

Levenshtein, Vladimir I. "Binary codes capable of correcting deletions, insertions and reversals." Soviet physics doklady. Vol. 10. 1966.

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

```
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:

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

where n is the length of the permutations x and y, and scaling factor $r = (n^2 - 1)/2$ (if n is odd) or $r = ((n^2)/2$ (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 Spearmans 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

Schiavinotto, Tommaso, and Thomas Stuetzle. "A review of metrics on permutations for search landscape analysis." Computers & operations research 34.10 (2007): 3143-3153.

Reeves, Colin R. "Landscapes, operators and heuristic search." Annals of Operations Research 86 (1999): 473-490.

Examples

```
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 Spearmans 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

Schiavinotto, Tommaso, and Thomas Stuetzle. "A review of metrics on permutations for search landscape analysis." Computers & operations research 34.10 (2007): 3143-3153.

Reeves, Colin R. "Landscapes, operators and heuristic search." Annals of Operations Research 86 (1999): 473-490.

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

```
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

Sevaux, Marc, and Kenneth Soerensen. "Permutation distance measures for memetic algorithms with population management." Proceedings of 6th Metaheuristics International Conference (MIC'05). 2005.

Reeves, Colin R. "Landscapes, operators and heuristic search." Annals of Operations Research 86 (1999): 473-490.

Examples

```
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. Note: In v2.4.0 of CEGO and earlier, this function actually computed the swap distance on the inverted permutations (i.e., on the rankings, rather than orderin). This is now (v2.4.2 and later) corrected by inverting the permutations x and y before computing the distance (ie. computing ordering first). The original behavior can be reproduced by `distancePermutationSwapInv`. This issue was kindly reported by Manuel Lopez-Ibanez and the difference in terms of behavior is discussed by Ekhine Irurozki and him (2021).

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

Schiavinotto, Tommaso, and Thomas Stuetzle. "A review of metrics on permutations for search landscape analysis." Computers & operations research 34.10 (2007): 3143-3153.

Irurozki, Ekhine and Ibanez-Lopez Unbalanced Mallows Models for Optimizing Expensive Black-Box Permutation Problems. In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2021. ACM Press, New York, NY, 2021. doi: 10.1145/3449639.3459366

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)
```

distancePermutationSwapInv *Inverse-Swap-Distance for Permutations*

Description

The swap distance on the inverse of permutations x and y. See distancePermutationSwap for non-inversed version.

Usage

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

distanceRealEuclidean *Euclidean Distance*

Description

The Euclidean distance for real vectors.

Usage

```
distanceRealEuclidean(x, y)
```

Arguments

x	first real vector
y	second real vector

Value

numeric distance value

$$d(x, y)$$

Examples

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

---

distanceSequenceLevenshtein

Levenshtein Distance forsSequences of numbers

Description

Levenshtein distance for two sequences of numbers

Usage

```
distanceSequenceLevenshtein(x, y)
```

Arguments

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

Value

numeric distance value

$$d(x, y)$$

Examples

```
#e.g., used for distance between integer sequence
x <- c(0,1,10,2,4)
y <- c(10,1,0,4,-4)
distanceSequenceLevenshtein(x,y)
p <- replicate(10,sample(1:5,3,replace=TRUE),simplify=FALSE)
distanceMatrix(p,distanceSequenceLevenshtein)
```

distanceStringHamming *Hamming Distance for Strings*

Description

Number of unequal letters in two strings.

Usage

```
distanceStringHamming(x, y)
```

Arguments

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

Value

numeric distance value

$$d(x, y)$$

Examples

```
distanceStringHamming("ABCD","AACC")
```

distanceStringLCStr *Longest Common Substring distance*

Description

Distance between strings, based on the longest common substring.

Usage

```
distanceStringLCStr(x, y)
```

Arguments

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

Value

numeric distance value

$$d(x, y)$$

Examples

```
distanceStringLCStr("ABCD","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

Value

A numerical vector of distances

Examples

```
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

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

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

Arguments

`X`	a symmetric matrix
`method`	a string, specifiying 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`	torelance value. Eigenvalues between `-tol` and `tol` are assumed to be zero.
	Symmetric, CNSD matrices are, e.g., euclidean distance matrices, whiche 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

Ikramov, K. and Savel'eva, N. Conditionally definite matrices, Journal of Mathematical Sciences, Kluwer Academic Publishers-Plenum Publishers, 2000, 98, 1-50

Glunt, W.; Hayden, T. L.; Hong, S. and Wells, J. An alternating projection algorithm for computing the nearest Euclidean distance matrix, SIAM Journal on Matrix Analysis and Applications, SIAM, 1990, 11, 589-600

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`	torelance value. Eigenvalues between `-tol` and `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`	torelance 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

```
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.

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 Testing 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
```

```
set.seed(seed)
#landscape
lF <- landscapeGeneratorUNI(cF(),dF)
x <- as.list(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)
plot(x,lF(x),type="l")
## glg landscape
#landscape
lF <- landscapeGeneratorGaussian(nGaussian=20,theta=1,
ratio=0.3,seed=seed,dF,cF)
plot(x,lF(x),type="l")
```

`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

```
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 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

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

```
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

```
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)
```

`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.

`distanceParametersUpper` 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.

`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`.

`scaling` If TRUE: Distances values are divided by maximum distance to avoid scale bias.

`reinterpolate` If TRUE: reinterpolation is used to generate better uncertainty estimates in the presence of noise.

`combineDistances` 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.

`userParameters` 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.

`indefiniteMethod` 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).

`indefiniteType` 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.

`indefiniteRepair` 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.

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 (see control parameter) and determined parameters for the model:

`theta` parameters of the kernel / correlation function determined with MLE.

`lambda` regularization constant (nugget) lambda

`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

`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 Surrogate Modelling - A Practical Guide. John Wiley & Sons.

Zaefferer, Martin; Stork, Joerg; Friese, Martina; Fischbach, Andreas; Naujoks, Boris; Bartz-Beielstein, Thomas. (2014). Efficient global optimization for combinatorial problems. In Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO '14). ACM, New York, NY, USA, 871-878. DOI=10.1145/2576768.2598282

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

Zaefferer, Martin and Bartz-Beielstein, Thomas (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

See Also

`predict.modelKriging`

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[-(1:15)]
x <- x[1:15]
# Determin 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))
```

`modelKrigingClust` *Build clustered model*

Description

This function builds an ensemble of Gaussian Process model, where each individual model is fitted to a partition of the parameter space. Partitions are generated by.

Usage

```
modelKrigingClust(x, y, distanceFunction, control = list())
```

Arguments

`x`	x
`y`	y
`distanceFunction`	distanceFunction
`control`	control

Value

an object

`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

Zaefferer, Martin; Stork, Joerg; Friese, Martina; Fischbach, Andreas; Naujoks, Boris; Bartz-Beielstein, Thomas. (2014). Efficient global optimization for combinatorial problems. In Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO '14). ACM, New York, NY, USA, 871-878. DOI=10.1145/2576768.2598282

See Also

`predict.modelLinear`

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[-(1:15)]
x <- x[1:15]
#determin 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)
```

---

modelRBFN *RBFN Model*

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 distance 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 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

Moraglio, Alberto, and Ahmed Kattan. "Geometric generalisation of surrogate model based optimisation to combinatorial spaces." Evolutionary Computation in Combinatorial Optimization. Springer Berlin Heidelberg, 2011. 142-154.

See Also

`predict.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[-(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 positition 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

```
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

```
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.

Value

mutated population

`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

---

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

```
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`).

See Also

optimEA, recombinationSelfAdapt

Examples

```
seed=0
N=5
require(ParamHelpers)
#distance
dF <- distancePermutationHamming
#mutation
mFs <- c(mutationPermutationSwap,mutationPermutationInterchange,
mutationPermutationInsert,mutationPermutationReversal)
rFs <- c(recombinationPermutationCycleCrossover,recombinationPermutationOrderCrossover1,
recombinationPermutationPositionBased,recombinationPermutationAlternatingPosition)
mF <- mutationSelfAdapt
selfAdaptiveParameters <- makeParamSet(
```

```
makeNumericParam("mutationRate", lower=1/N,upper=1, default=1/N),
makeDiscreteParam("mutationOperator", values=1:4, default=expression(sample(4,1))),
#1: swap, 2: interchange, 3: insert, 4: reversal mutation
makeDiscreteParam("recombinationOperator", values=1:4, default=expression(sample(4,1)))
#1: CycleX, 2: OrderX, 3: PositionX, 4: AlternatingPosition
)
#recombination
rF <-  recombinationSelfAdapt
#creation
cF <- function()sample(N)
#objective function
lF <- landscapeGeneratorUNI(1:N,dF)
#start optimization
set.seed(seed)
res <- optimEA(,lF,list(parameters=list(mutationFunctions=mFs,recombinationFunctions=rFs),
creationFunction=cF,mutationFunction=mF,recombinationFunction=rF,
popsize=15,budget=100,targetY=0,verbosity=1,selfAdaption=selfAdaptiveParameters,
vectorized=TRUE)) ##target function is "vectorized", expects list as input
res$xbest
```

`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

```
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 torelance value. Eigenvalues between -tol and tol are assumed to be zero.
conv.tol	convergence torelance 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

list with:

mat nearestCNSD 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

Glunt, W.; Hayden, T. L.; Hong, S. and Wells, J. An alternating projection algorithm for computing the nearest Euclidean distance matrix, SIAM Journal on Matrix Analysis and Applications, SIAM, 1990, 11, 589-600

See Also

nearPD, correctionCNSD, correctionDistanceMatrix

Examples

```
# 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

```
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

Wikipedia contributors. "2-opt." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 13 Jun. 2014. Web. 21 Oct. 2014.

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
```

---

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), 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 doe 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(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$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

Zaefferer, Martin; Stork, Joerg; Friese, Martina; Fischbach, Andreas; Naujoks, Boris; Bartz-Beielstein, Thomas. (2014). Efficient global optimization for combinatorial problems. In Proceedings of the 2014 conference on Genetic and evolutionary computation (GECCO '14). ACM, New York, NY, USA, 871-878. DOI=10.1145/2576768.2598282

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

See Also

`modelKriging`, `modelLinear`, `modelRBFN`, `buildModel`, `optimEA`

Examples

```
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

```
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). Default 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: `recombinationPermutation` which is cycle crossover for permutations.

`recombinationRate` Number of offspring, defined by the fraction of the population (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 control 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 numerical/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

```
#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
cntrl <- list(
budget = 100,
popsize = 5,
creationFunction = cf,
vectorized = FALSE, #set to TRUE if f evaluates a list of individuals
recombinationFunction = recombinationBinary2Point,
recombinationRate = 0.1,
mutationFunction = mutationBinaryBitFlip,
parameters=list(mutationRate = 1/N),
archive=FALSE #recommended for larger budgets. do not change.
```

```
)
#start algorithm
set.seed(1)
res <- optimEA(fun=f,control=cntrl)
res$xbest
res$ybest
```

---

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

```
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)

popsize 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., reltol).

`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_N
"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","NLO

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

```
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
```

---

optimMIES *Mixed Integer Evolution Strategy (MIES)*

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 Ta,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

Rui Li, Michael T. M. Emmerich, Jeroen Eggermont, Thomas Baeck, Martin Schuetz, Jouke Dijkstra, and Johan H. C. Reiber. 2013. Mixed integer evolution strategies for parameter optimization. Evol. Comput. 21, 1 (March 2013), 29-64.

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)
objFun <- function(x){
x[[3]] <- round(x[[3]])
x[[4]] <- round(x[[4]])
y <- sum(as.numeric(x[1:4])^2)
if(x[[5]]==1 & x[[6]]==4)
y <- exp(y)
else
y <- y^2
if(x[[5]]==3)
y<-y-1
if(x[[5]]==5)
y<-y-2
if(x[[6]]==1)
y<-y*2
if(x[[6]]==2)
y<-y * 1.54
```

```
if(x[[6]]==3)
y<- y +2
if(x[[6]]==4)
y<- y * 0.5
if(x[[5]]==1)
y<- y * 9
y
}
res <- optimMIES(,objFun,controlList)
res$xbest
res$ybest
```

---

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)
	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 *Kriging 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$s (standard deviation)
FALSE:object$yonly

See Also

modelKriging

simulate.modelKriging

predict.modelKrigingClust *Clustered Kriging Prediction*

Description

Predict with a model fit resulting from `modelKrigingClust`.

Usage

```
## S3 method for class 'modelKrigingClust'
predict(object, newdata, ...)
```

Arguments

`object`	fit of the clustered Kriging model ensemble (settings and parameters), of class `modelKrigingClust`.
`newdata`	list of samples to be predicted
`...`	further arguments, currently not used

Value

list with function value (mean) `object$y` and uncertainty estimate `object$s` (standard deviation)

See Also

`predict.modelKriging`

predict.modelLinear *Predict: Combinatorial Kriging*

Description

Predict with amodelLinear fit.

Usage

```
## 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

```
## 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 `$s` (standard deviation)
FALSE:$yonly

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

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

Arguments

population	List of bit-strings
parameters	not used

Value

population of recombined offspring

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

References

Martin Zaefferer and Thomas Bartz-Beielstein. (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

See Also

correctionDefinite, correctionDistanceMatrix, correctionKernelMatrix, correctionCNSD, repairConditionsDistanceMatrix

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)
K <- exp(-0.01*D)
K <- correctionDefinite(K,type="PSD")$mat
K
K <- repairConditionsCorrelationMatrix(K)
```

repairConditionsDistanceMatrix
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-definitene (CNSD), symmetric) are assumed to be given.

Usage

```
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

Martin Zaefferer and Thomas Bartz-Beielstein. (2016). Efficient Global Optimization with Indefinite Kernels. Parallel Problem Solving from Nature-PPSN XIV. Accepted, in press. Springer.

See Also

correctionDefinite, correctionDistanceMatrix, correctionKernelMatrix, correctionCNSD, repairConditionsCorrelationMatrix

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)
D <- correctionCNSD(D)
D
D <- repairConditionsDistanceMatrix(D)
D
```

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

```
## 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

Value

Returned value depends on the setting of object$simulationReturnAll

References

N. A. Cressie. Statistics for Spatial Data. JOHN WILEY & SONS INC, 1993.

C. Lantuejoul. Geostatistical Simulation - Models and Algorithms. Springer-Verlag Berlin Heidelberg, 2002.

See Also

modelKriging, predict.modelKriging

`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

`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

```
solutionFunctionGeneratorPermutation(N)
```

Arguments

N	length of the permutations returned

Value

returns a function, without any arguments

Examples

```
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

```
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

```
fun <- solutionFunctionGeneratorString(10,c("A","C","G","T"))
fun()
fun()
fun()
```

testFunctionGeneratorSim *Simulation-based Test Function Generator, Data Interface*

Description

Generate test functions for assessment of optimization algorithms with non-conditional or conditional simulation, based on real-world data.

Usage

```
testFunctionGeneratorSim(
  x,
  y,
  xsim,
  distanceFunction,
  controlModel = list(),
  controlSimulation = list()
)
```

Arguments

x	list of samples in input space, training data
y	column vector of observations for each sample, training data
xsim	list of samples in input space, for simulation
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.
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

N. A. Cressie. Statistics for Spatial Data. JOHN WILEY & SONS INC, 1993.

C. Lantuejoul. Geostatistical Simulation - Models and Algorithms. Springer-Verlag Berlin Heidelberg, 2002.

Zaefferer, M.; Fischbach, A.; Naujoks, B. & Bartz-Beielstein, T. Simulation Based Test Functions for Optimization Algorithms Proceedings of the Genetic and Evolutionary Computation Conference 2017, ACM, 2017, 8.

See Also

`modelKriging`, `simulate.modelKriging`, `createSimulatedTestFunction`,

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)}
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