Package 'xegaDfGene'

February 16, 2026

Title Gene Operations for Real-Coded Genes

Version 1.0.0.9

Description Representation-dependent gene-level operations
for genetic and evolutionary algorithms with real-coded genes
used in the R-package 'xega' <https://CRAN.R-project.org/package=xega>
are collected in this package. The common feature of the gene
operations is that all of them are useful for derivation-free
optimization algorithms. At the moment the package
implements initialization, mutation, crossover, and replication
operations for differential evolution as described in
Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005)
<doi:10.1007/3-540-31306-0>. In addition, several (more recent)
methods for determining the scale factor are provided.
For 'xega''s architecture,
see Geyer-Schulz, A. (2025) <doi:10.5445/IR/1000187255>.

License MIT + file LICENSE

URL https://github.com/ageyerschulz/xegaDfGene

Encoding UTF-8

RoxygenNote 7.3.3

Suggests testthat (>= 3.0.0)

Imports rlang, xegaSelectGene

NeedsCompilation no

Author Andreas Geyer-Schulz [aut, cre] (ORCID:
<https://orcid.org/0009-0000-5237-3579>)

Maintainer Andreas Geyer-Schulz <Andreas.Geyer-Schulz@kit.edu>

Repository CRAN

Date/Publication 2026-02-16 14:30:08 UTC

Contents

`CauchySF` *Cauchy distribution based scale factor.*

Description

The scale factors is Cauchy distributed with

1. location parameter `1-0.6*t/T` and
2. scale parameter `0.7-0.4(1-t/T))`.

Usage

```
CauchySF(lF)
```

Arguments

`lF`	Local configuration.

Details

The parameters are constant functions defined in `lF`:

1. t is the current iteration (`lF$cGeneration`).
2. T is the total number of generations (`lF$Generations`).

The scale factor is bounded from above by `1`. For `SF<0`, The scale factor is set to `abs(rnorm(1, 0, 0.2))`.

For details, see section 3 of Sharma et al. (2019), pp. 929-931 or Fan et al. (2017), pp. 6844-6845.

Value

A scale factor.

References

Sharma, Prashant; Sharma, Harish; Kumar, Sandeep; Bansal, Jagdish Chand (2019): A Review on Scale Factor Strategies in Differential Evolution Algorithm. pp. 925-934. In: Bansal, Jagdish Chand et al. (2019) Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing, Vol. 817. Springer, Singapore, 2019. (ISBN:978-981-13-1594-7)

Fan, Qinqin; Yan, Xuefeng; Xue, Yu (2017) Prior knowledge guided differential evolution, Soft Computing 21(22), 6841 - 6858. (doi:10.1007/s00500-016-2235-6)

See Also

Other Scale Factor: `ConstScaleFactor()`, `DETVSF()`, `FitnessBasedSelfAdaptiveSF()`, `RandomGaussianSF()`, `UniformRandomScaleFactor()`, `UniformRandomScaleFactorDERSF()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
lF$Generations<-parm(4)
lF$cGeneration<-parm(0)
CauchySF(lF)
lF$cGeneration<-parm(1)
CauchySF(lF)
lF$cGeneration<-parm(2)
CauchySF(lF)
lF$cGeneration<-parm(3)
CauchySF(lF)
lF$cGeneration<-parm(4)
CauchySF(lF)
```

ConstScaleFactor *Constant scale factor for differential evolution.*

Description

Constant scale factor for differential evolution.

Usage

```
ConstScaleFactor(lF)
```

Arguments

lF	Local configuration.

Value

A constant scale factor.

See Also

Other Scale Factor: CauchySF(), DETVSF(), FitnessBasedSelfAdaptiveSF(), RandomGaussianSF(), UniformRandomScaleFactor(), UniformRandomScaleFactorDERSF()

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
lF$ScaleFactor1<-parm(0.90)
ConstScaleFactor(lF)
lF$ScaleFactor1<-parm(1.10)
ConstScaleFactor(lF)
```

DETVSF *Scale factor with time dependent linear decay.*

Description

The scale factor is linear decaying from an upper bound to a lower bound with the number of generations. The scale factor is computed by UB-(UB-LB)*(t/T). See section 16.2 of Sharma et al. (2019), p.941 and Das et al. (2005).

Usage

```
DETVSF(lF)
```

Arguments

lF	Local configuration.

Details

The parameters are constant functions defined in lF:

1. UB is the upper bound of the scale factor (lF$ScaleFactor1).
2. LB is lower upper bound of the scale factor (lF$ScaleFactor2).
3. t is the current iteration (lF$cGeneration).
4. T is the total number of generations (lF$Generations).

A simple check shows that the formulas given in both papers are not correct.

Value

A scale factor.

References

Sharma, Prashant; Sharma, Harish; Kumar, Sandeep; Bansal, Jagdish Chand (2019): A Review on Scale Factor Strategies in Differential Evolution Algorithm. pp. 925-934. In: Bansal, Jagdish Chand et al. (2019) Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing, Vol. 817. Springer, Singapore, 2019. (ISBN:978-981-13-1594-7)

Das, Swagatam; Konar, Amit; Chakraborty, Uday K. (2005): Two Improved Differential Evolution Schemes for Faster Global Search. pp. 991-998. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, Association for Computing Machinery, New York. (doi:10.1145/1068009.1068177)

See Also

Other Scale Factor: CauchySF(), ConstScaleFactor(), FitnessBasedSelfAdaptiveSF(), RandomGaussianSF(), UniformRandomScaleFactor(), UniformRandomScaleFactorDERSF()

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
lF$ScaleFactor1<-parm(1.10)
lF$ScaleFactor2<-parm(0.5)
lF$Generations<-parm(4)
lF$cGeneration<-parm(0)
DETVSF(lF)
lF$cGeneration<-parm(1)
DETVSF(lF)
lF$cGeneration<-parm(2)
DETVSF(lF)
lF$cGeneration<-parm(3)
DETVSF(lF)
lF$cGeneration<-parm(4)
```

```
DETVSF(lF)
```

`FitnessBasedSelfAdaptiveSF` *Fitness based self adaptive scale factor.*

Description

The scale factor is a product of a relative fitness based term times a uniform random number.

Usage

```
FitnessBasedSelfAdaptiveSF(lF)
```

Arguments

`lF`	Local configuration.

Details

The parameters are:

1. fit: fitness of gene0.
2. max fit: the best fitness given by lF$CBestFitness().

The scale factor is in the interval of `[-1.05, 1.05]`.

For details, see section 4 of Sharma et al. (2019), p. 931 or Sharma et al. (2013).

If the optimal fitness value is `0`, the quotient for computing the relative fitness is `Inf` or `-Inf`. In this case, `SF=0.1`.

Value

A scale factor.

References

Sharma, Prashant; Sharma, Harish; Kumar, Sandeep; Bansal, Jagdish Chand (2019): A Review on Scale Factor Strategies in Differential Evolution Algorithm. pp. 925-934. In: Bansal, Jagdish Chand et al. (2019) Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing, Vol. 817. Springer, Singapore, 2019. (ISBN:978-981-13-1594-7)

Sharma, Harish; Shrivastava, Pragati; Bansal, Jagdish Chand; Tiwari, Ritu (2013) Fitness Based Self Adaptive Differential Evolution pp. 71-94. In: Terrazas et al. (2013) Nature Inspired Cooperative Strategies for Optimization (NICSO 2013) Springer, Heidelberg. (doi:10.1007/978-3-319-01692-4_6)

See Also

Other Scale Factor: CauchySF(), ConstScaleFactor(), DETVSF(), RandomGaussianSF(), UniformRandomScaleFactor() UniformRandomScaleFactorDERSF()

Examples

```
parm<-function(x){function() {return(x)}}
lFxegaDfGene$CBestFitness<-parm(35)
pop3<-lapply(rep(0,3), function(x) xegaDfGene::xegaDfInitGene(lFxegaDfGene))
lFxegaDfGene$gene0<-pop3[[1]]
FitnessBasedSelfAdaptiveSF(lFxegaDfGene)
lFxegaDfGene$gene0<-pop3[[2]]
FitnessBasedSelfAdaptiveSF(lFxegaDfGene)
lFxegaDfGene$gene0<-pop3[[3]]
FitnessBasedSelfAdaptiveSF(lFxegaDfGene)
```

---

lFxegaDfGene *Generate local functions and objects*

Description

lFxegaDfGene() is the list of functions containing a definition of all local objects required for the use of evaluation functions. We reference this object as local configuration. When adding additional functions, this list must be extended by the constant (functions) needed to configure them.

Usage

```
lFxegaDfGene
```

Format

An object of class list of length 32.

Details

We use the local configuration for:

1. Replacing all constants with constant functions.
   Rationale: We need one formal argument (the local function list lF) and we can dispatch multiple functions. E.g., lF$verbose().
2. Dynamically binding a local function with a definition from a proper function factory. E.g., the selection methods lF$SelectGene() and lF$SelectMate().
3. Gene representations requiring special functions to handle them: lF$InitGene(), lF$DecodeGene(), lF$EvalGene(), lF$ReplicateGene(), ...

See Also

Other Configuration: xegaDfCrossoverFactory(), xegaDfGeneMapFactory(), xegaDfMutationFactory(), xegaDfReplicationFactory(), xegaDfScaleFactorFactory()

RandomGaussianSF *Random Gaussian scale factor (RGSF).*

Description

The scale factor is drawn randomly from one of the two Gaussian distributions `abs(rnorm(1, 0.3, 0.3))` or `abs(rnorm(1, 0.7, 0.3))`.

Usage

```
RandomGaussianSF(lF)
```

Arguments

`lF`	Local configuration.

Details

For details, see section 5 of Sharma et al. (2019), p. 932 or Li and Yin (2016), p. 555. Note that the range given in both papers is false.

Value

A scale factor.

References

Li, Xiangtao AND Yin, Minghao (2016) Modified differential evolution with self-adaptive parameters method. Journal of Combinatorial Optimization 31(2), 546-576. (doi:10.1007/s10878-014-9773-6)

See Also

Other Scale Factor: `CauchySF()`, `ConstScaleFactor()`, `DETVSF()`, `FitnessBasedSelfAdaptiveSF()`, `UniformRandomScaleFactor()`, `UniformRandomScaleFactorDERSF()`

Examples

```
RandomGaussianSF(lFxegaDfGene)
RandomGaussianSF(lFxegaDfGene)
RandomGaussianSF(lFxegaDfGene)
```

UniformRandomScaleFactor

Uniform random scale factor for differential evolution.

Description

The scale factor is drawn from `0.000001` to `1.0`.

Usage

```
UniformRandomScaleFactor(lF)
```

Arguments

`lF`	Local configuration.

Value

A constant scale factor.

See Also

Other Scale Factor: `CauchySF()`, `ConstScaleFactor()`, `DETVSF()`, `FitnessBasedSelfAdaptiveSF()`, `RandomGaussianSF()`, `UniformRandomScaleFactorDERSF()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
UniformRandomScaleFactor(lF)
UniformRandomScaleFactor(lF)
```

UniformRandomScaleFactorDERSF

Restricted uniform random scale factor for differential evolution (DERSF).

Description

The scale factor is computed by `0.5*(1+rand(0,1))`. See section 16.1 of Sharma et al. (2019), p.940 and Das et al. (2005). The mean value of the `SF` is `0.75`.

Usage

```
UniformRandomScaleFactorDERSF(lF)
```

Arguments

lF	Local configuration.

Value

A constant scale factor.

References

Sharma, Prashant; Sharma, Harish; Kumar, Sandeep; Bansal, Jagdish Chand (2019): A Review on Scale Factor Strategies in Differential Evolution Algorithm. pp. 925-934. In: Bansal, Jagdish Chand et al. (2019) Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing, Vol. 817. Springer, Singapore, 2019. (ISBN:978-981-13-1594-7)

Das, Swagatam; Konar, Amit; Chakraborty, Uday K. (2005): Two Improved Differential Evolution Schemes for Faster Global Search. pp. 991-998. In: Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, Association for Computing Machinery, New York. (doi:10.1145/1068009.1068177)

See Also

Other Scale Factor: `CauchySF()`, `ConstScaleFactor()`, `DETVSF()`, `FitnessBasedSelfAdaptiveSF()`, `RandomGaussianSF()`, `UniformRandomScaleFactor()`

Examples

```
parm<-function(x){function() {return(x)}}
lF<-list()
lF$ScaleFactor1<-parm(0.90)
UniformRandomScaleFactorDERSF(lF)
lF$ScaleFactor1<-parm(1.10)
UniformRandomScaleFactorDERSF(lF)
```

---

xegaDfCrossGene *One point crossover of 2 genes.*

Description

`xegaDfCrossGene()` randomly determines a cut point. It combines the parameters before the cut point of the first gene with the parameters after the cut point from the second gene (kid 1).

Usage

```
xegaDfCrossGene(gg1, gg2, lF)
```

Arguments

gg1	Real-coded gene.
gg2	Real-coded gene.
lF	Local configuration of the genetic algorithm.

Value

Real-coded gene.

See Also

Other Crossover (Returns 1 Kid): `xegaDfUCrossGene()`, `xegaDfUPCrossGene()`

Examples

```
gene1<-xegaDfInitGene(lFxegaDfGene)
gene2<-xegaDfInitGene(lFxegaDfGene)
gene3<-xegaDfCrossGene(gene1, gene2, lFxegaDfGene)
```

---

xegaDfCrossoverFactory

Configure the crossover function of a genetic algorithm.

Description

`xegaDfCrossoverFactory()` implements the selection of one of the crossover functions in this package by specifying a text string. The selection fails ungracefully (produces a runtime error) if the label does not match. The functions are specified locally.

Current support:

Crossover functions with one kid:

1. "CrossGene" returns `CrossGene()`.
2. "UCrossGene" returns `UCrossGene()`. Default.
3. "UPCrossGene" returns `UPCrossGene()`.

Usage

```
xegaDfCrossoverFactory(method = "UCrossGene")
```

Arguments

`method`	A string specifying the crossover function.

Details

All crossover operations return a 1 kid. This implies that some part of the genetic material is lost.

Value

Crossover function for genes.

See Also

Other Configuration: lFxegaDfGene, xegaDfGeneMapFactory(), xegaDfMutationFactory(), xegaDfReplicationFacto
xegaDfScaleFactorFactory()

Examples

```
XGene<-xegaDfCrossoverFactory("UCrossGene")
gene1<-xegaDfInitGene(lFxegaDfGene)
gene2<-xegaDfInitGene(lFxegaDfGene)
XGene(gene1, gene2, lFxegaDfGene)
```

---

xegaDfDecodeGene *Decode a gene*

Description

xegaDfDecodeGene() decodes a real gene.

Usage

```
xegaDfDecodeGene(gene, lF)
```

Arguments

gene	Real gene.
lF	Local configuration of the genetic algorithm.

Details

xegaDfDecodeGene() is the identity function.

Value

Decoded gene.

See Also

Other Decoder: xegaDfGeneMapIdentity()

Examples

```
gene<-xegaDfInitGene(lFxegaDfGene)
xegaDfDecodeGene(gene, lFxegaDfGene)
```

xegaDfGene *Package xegaDfGene.*

Description

Genetic operations for real-coded genetic and evolutionary algorithms.

Details

For real-coded genes, the xegaDfGene package provides

- Gene initialization.
- Decoding of parameters.
- Scaling functions as a function factory for configuration.
- Mutation functions as well as a function factory for configuration.
- Crossover functions as well as a function factory for configuration.
- Replication functions as well as a function factory for configuration.

Current support: Functions for differential evolution (de). See Price et al. (2005).

Real-Coded Gene Representation

A real-coded gene is a named list:

- $gene1: the gene must be a vector of reals.
- $fit: the fitness value of the gene (for EvalGeneDet and EvalGeneU) or the mean fitness (for stochastic functions evaluated with EvalGeneStoch).
- $evaluated: has the gene been evaluated?
- $evalFail: has the evaluation of the gene failed?
- $var: the cumulative variance of the fitness of all evaluations of a gene. (For stochastic functions)
- $sigma: the standard deviation of the fitness of all evaluations of a gene. (For stochastic functions)
- $obs: the number of evaluations of a gene. (For stochastic functions)

Abstract Interface of Problem Environment

We reuse the abstract interface of a problem environment for binary-coded genes. The number of parameters is given by length(penv$bitlength()).

A problem environment penv must provide:

- $f(parameters, gene, lF): Function with a real parameter vector as first argument which returns a gene with evaluated fitness.
- $genelength(): The number of bits of the binary coded real parameter vector. Used in InitGene.

- $bitlength(): A vector specifying the number of bits used for coding each real parameter. If `penv$bitlength()[1]` is `20`, then `parameters[1]` is coded by 20 bits. Used in `GeneMap`.
- $lb(): The lower bound vector of each parameter. Used in `GeneMap`.
- $ub(): The upper bound vector of each parameter. Used in `GeneMap`.

The Architecture of the xegaX-Packages

The xegaX-packages are a family of R-packages which implement eXtended Evolutionary and Genetic Algorithms (xega). The architecture has 3 layers, namely the user interface layer, the population layer, and the gene layer:

- The user interface layer (package `xega`) provides a function call interface and configuration support for several algorithms: genetic algorithms (sga), permutation-based genetic algorithms (sgPerm), derivation-free algorithms as e.g. differential evolution (sgde), grammar-based genetic programming (sgp) and grammatical evolution (sge).
- The population layer (package `xegaPopulation`) contains population-related functionality as well as support for population statistics dependent adaptive mechanisms and parallelization.
- The gene layer is split into a representation-independent and a representation-dependent part:
  1. The representation-independent part (package `xegaSelectGene`) is responsible for variants of selection operators, evaluation strategies for genes, as well as profiling and timing capabilities.
  2. The representation-dependent part consists of the following packages:
     - `xegaGaGene` for binary coded genetic algorithms.
     - `xegaPermGene` for permutation-based genetic algorithms.
     - `xegaDfGene` for derivation-free algorithms as e.g. differential evolution.
     - `xegaGpGene` for grammar-based genetic algorithms.
     - `xegaGeGene` for grammatical evolution algorithms.

     The packages `xegaDerivationTrees` and `xegaBNF` support the last two packages: `xegaBNF` essentially provides a grammar compiler, and `xegaDerivationTrees` is an abstract data type for derivation trees.

Copyright

License

<URL

https://github.com/ageyerschulz/xegaDfGene>

Installation

From CRAN by `install.packages('xegaDfGene')`

Author(s)

Andreas Geyer-Schulz

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Useful links:

- https://github.com/ageyerschulz/xegaDfGene

---

xegaDfGeneMapFactory *Configure the gene map function of a genetic algorithm.*

Description

`xegaDfGeneMapFactory()` implements the selection of one of the GeneMap functions in this package by specifying a text string. The selection fails ungracefully (produces a runtime error) if the label does not match. The functions are specified locally.

Current support:

1. "Identity" returns `GeneMapIdentity()`. (Default)

Usage

```
xegaDfGeneMapFactory(method = "Identity")
```

Arguments

`method`	String specifying the GeneMap function.

Value

Gene map function for genes.

See Also

Other Configuration: `lFxegaDfGene`, `xegaDfCrossoverFactory()`, `xegaDfMutationFactory()`, `xegaDfReplicationFactory()`, `xegaDfScaleFactorFactory()`

Examples

```
XGene<-xegaDfGeneMapFactory("Identity")
gene1<-xegaDfInitGene(lFxegaDfGene)
XGene(gene1, lFxegaDfGene$penv)
```

xegaDfGeneMapIdentity *Map the parameter vector of a real-coded gene to an identical vector.*

Description

`xegaDfGeneMapIdentity()` maps the real parameter vector to an identical vector.

Usage

```
xegaDfGeneMapIdentity(gene, penv)
```

Arguments

`gene`	Real-coded gene (the genotype).
`penv`	Problem environment.

Details

A *gene* is a list with

1. `$evaluated`: Boolean. TRUE if the fitness is known.
2. `$fit`: The fitness of the genotype of `$gene1`.
3. `$gene1`: A real parameter vector (the genotopye).

Value

Decoded gene (the phenotype).

See Also

Other Decoder: `xegaDfDecodeGene()`

Examples

```
gene<-xegaDfInitGene(lFxegaDfGene)
xegaDfGeneMapIdentity(gene$gene1, lFxegaDfGene$penv)
```

`xegaDfInitGene` *Initialize a real-coded gene.*

Description

`xegaDfInitGene()` generates a random real-coded gene with a given length. The length of the gene is the length of `lF$penv$bitlength()`.

Usage

```
xegaDfInitGene(lF)
```

Arguments

`lF`	Local configuration of the genetic algorithm.

Details

In the real-coded representation of package `xegaDf`, *gene* is at least a list with

1. `$evaluated`: Boolean. TRUE if the fitness is known.
2. `$fit`: The fitness of the genotype of `$gene1`.
3. `$gene1`: a real vector (the genotype).

Value

A real-coded gene (a named list):

- `$evaluated`: FALSE. See package `xegaSelectGene`.
- `$evalFail`: FALSE. Set by the error handler(s) of the Evaluation Functions in package `xegaSelectGene` in the case of failure.
- `$fit`: Fitness.
- `$gene1`: A vector of reals.

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Other Intialization: `xegaGedeInitGene()`

Examples

```
xegaDfInitGene(lFxegaDfGene)
```

`xegaDfMutateGeneDE` *Mutate a gene (differential mutation).*

Description

`xegaDfMutateGeneDE()` mutates a real-coded gene. The scale factor is given by `lF$ScaleFactor()`.

Usage

```
xegaDfMutateGeneDE(gene0, gene1, gene2, lF)
```

Arguments

`gene0`	Real-coded gene (the base vector).
`gene1`	Real-coded gene.
`gene2`	Real-coded gene.
`lF`	Local configuration.

Details

The difference between gene1 and gene2 is scaled by `lF$ScaleFactor()` and added to gene0.

Value

Real-coded gene.

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

Examples

```
gene0<-xegaDfInitGene(lFxegaDfGene)
gene1<-xegaDfInitGene(lFxegaDfGene)
gene2<-xegaDfInitGene(lFxegaDfGene)
gene<-xegaDfMutateGeneDE(gene0, gene1, gene2, lFxegaDfGene)
```

xegaDfMutationFactory *Configure the mutation function of a genetic algorithm.*

Description

xegaDfMutationFactory() implements the selection of one of the mutation functions in this package by specifying a text string. The selection fails ungracefully (produces a runtime error), if the label does not match. The functions are specified locally.

Current support:

1. "MutateGene" returns xegaDfMutateGeneDE(). To provide a working default for more than one gene representation.
2. "MutateGeneDE" returns xegaDfMutateGeneDE().

Usage

```
xegaDfMutationFactory(method = "MutateGene")
```

Arguments

method	A string specifying the mutation function.

Value

A mutation function for genes.

See Also

Other Configuration: lFxegaDfGene, xegaDfCrossoverFactory(), xegaDfGeneMapFactory(), xegaDfReplicationFactory(), xegaDfScaleFactorFactory()

Examples

```
Mutate<-xegaDfMutationFactory("MutateGene")
gene1<-xegaDfInitGene(lFxegaDfGene)
gene2<-xegaDfInitGene(lFxegaDfGene)
gene3<-xegaDfInitGene(lFxegaDfGene)
Mutate(gene1, gene2, gene3, lFxegaDfGene)
```

`xegaDfReplicateGeneDE` *Replicates a gene (differential evolution).*

Description

`xegaDfReplicateGeneDE()` replicates a gene. Replication is the reproduction function which uses crossover and mutation. The control flow of differential evolution is as follows:

- A target gene is selected from the population.
- A mutant gene is generated by differential mutation.
- The gene and the mutant gene are crossed to get a new gene.
- The gene is accepted if it is at least as good as the target gene.

Usage

```
xegaDfReplicateGeneDE(pop, fit, lF)
```

Arguments

`pop`	Population of real-coded genes.
`fit`	Fitness vector.
`lF`	Local configuration of the genetic algorithm.

Details

For `selection="UniformP"`, for `crossover="UPCrossGene"` and for `accept="Best"` this is the algorithm of Price, Storn and Lampinen (2005), page 41.

To concentrate sampling around the best gene for faster convergence of global functions, configure `selection="TopK"` and around the best 3 genes, add `topK=3`.

targetGene is selected with `lF$SelectGene()`, gene0, gene1, and gene2 are selected by `lF$SelectMate()`.

Value

A list of one gene.

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Other Replication: `xegaDfReplicateGeneDEPipeline()`, `xegaDfReplicateGeneDEPipelineG()`

Examples

```
pop10<-lapply(rep(0,10), function(x) xegaDfGene::xegaDfInitGene(lFxegaDfGene))
epop10<-lapply(pop10, lFxegaDfGene$EvalGene, lF=lFxegaDfGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
newgenes<-xegaDfReplicateGeneDE(pop10, fit10, lFxegaDfGene)
```

`xegaDfReplicateGeneDEPipeline`

Generates a function closure with a gene pipeline for differential evolution.

Description

`xegaDfReplicateGeneDEPipeline()` embeds a gene pipeline for differential evolution and the genes necessary to evaluate it into a function closure. Replication is the reproduction function which uses crossover and mutation. The control flow of differential evolution is as follows:

- A target gene is selected from the population.
- A mutant gene is generated by differential mutation.
- The gene and the mutant gene are crossed to get a new gene.
- The gene is accepted if it is at least as good as the target gene.

The execution is shifted to the evaluation phase.

Usage

```
xegaDfReplicateGeneDEPipeline(pop, fit, lF)
```

Arguments

`pop`	Population of real-coded genes.
`fit`	Fitness vector.
`lF`	Local configuration of the genetic algorithm.

Details

For `selection="UniformP"`, for `crossover="UPCrossGene"` and for `accept="Best"` this is the algorithm of Price, Storn and Lampinen (2005), page 41.

To concentrate sampling around the best gene for faster convergence of global functions, configure `selection="TopK"` and around the best 3 genes, add `topK=3`.

`xegaDfReplicateGeneDEPipeline()` is a genetic operator constructor which generates a function closure which embeds differential mutation, crossover and the acceptance rule.

Value

A list of differential evolution pipelines.

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Other Replication: `xegaDfReplicateGeneDE()`, `xegaDfReplicateGeneDEPipelineG()`

Examples

```
pop10<-lapply(rep(0,10), function(x) xegaDfGene::xegaDfInitGene(lFxegaDfGene))
epop10<-lapply(pop10, lFxegaDfGene$EvalGene, lF=lFxegaDfGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
ng<-xegaDfReplicateGeneDEPipeline(pop10, fit10, lFxegaDfGene)
```

`xegaDfReplicateGeneDEPipelineG` *Generates a gene with an embedded gene pipeline for differential evolution.*

Description

`xegaDfReplicateGeneDEPipelineG()` embeds a gene pipeline for differential evolution and the genes necessary to evaluate it into a gene. Replication is the reproduction function which uses crossover and mutation. The control flow of differential evolution is as follows:

- A target gene is selected from the population.
- A mutant gene is generated by differential mutation.
- The gene and the mutant gene are crossed to get a new gene.
- The gene is accepted if it is at least as good as the target gene.

The execution is shifted to the evaluation phase.

Usage

```
xegaDfReplicateGeneDEPipelineG(pop, fit, lF)
```

Arguments

`pop`	Population of real-coded genes.
`fit`	Fitness vector.
`lF`	Local configuration of the genetic algorithm.

Details

For `selection="UniformP"`, for `crossover="UPCrossGene"` and for `accept="Best"` this is the algorithm of Price, Storn and Lampinen (2005), page 41.

To concentrate sampling around the best gene for faster convergence of global functions, configure `selection="TopK"` and around the best 3 genes, add `topK=3`.

`xegaDfReplicateGeneDEPipelineG()` is a genetic operator constructor which generates a gene with an embedded differential evolution operator pipeline (differential mutation, crossover and the acceptance rule).

Value

A gene with an embedded differential evolution pipeline.

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Other Replication: `xegaDfReplicateGeneDE()`, `xegaDfReplicateGeneDEPipeline()`

Examples

```
pop10<-lapply(rep(0,10), function(x) xegaDfGene::xegaDfInitGene(lFxegaDfGene))
epop10<-lapply(pop10, lFxegaDfGene$EvalGene, lF=lFxegaDfGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
ng<-xegaDfReplicateGeneDEPipelineG(pop10, fit10, lFxegaDfGene)
g<-ng[[1]]$Pipeline(ng[[1]], lFxegaDfGene)
```

`xegaDfReplicationFactory` *Configure the replication function of a genetic algorithm.*

Description

`xegaDfReplicationFactory()` implements the selection of a replication method.

Current support:

1. "DE" returns `xegaDfReplicateGeneDE()`.
2. "DEPipeline" returns `xegaDfReplicateGeneDEPipeline()`.
3. "DEPipelineG" returns `xegaDfReplicateGeneDEPipelineG()`.

Usage

```
xegaDfReplicationFactory(method = "DE")
```

Arguments

method	A string specifying the replication function.

Value

A replication function for genes.

See Also

Other Configuration: lFxegaDfGene, xegaDfCrossoverFactory(), xegaDfGeneMapFactory(), xegaDfMutationFactory(), xegaDfScaleFactorFactory()

Examples

```
pop10<-lapply(rep(0,10), function(x) xegaDfInitGene(lFxegaDfGene))
epop10<-lapply(pop10, lFxegaDfGene$EvalGene, lF=lFxegaDfGene)
fit10<-unlist(lapply(epop10, function(x) {x$fit}))
Replicate<-xegaDfReplicationFactory("DE")
newgenes2<-Replicate(pop10, fit10, lFxegaDfGene)
```

---

xegaDfScaleFactorFactory

Configure the scale factor function for differential mutation.

Description

xegaDfScaleFactorFactory() implements the selection of one of the scale factor functions in this package by specifying a text string. The selection fails ungracefully (produces a runtime error) if the label does not match. The functions are specified locally.

Current support:

1. "Const" returns ConstScaleFactor().
2. "Uniform" returns UniformRandomScaleFactor().
3. "DERSF" returns UniformRandomScaleFactorDERSF().
4. "DEVSF" returns DEVSF().
5. "CauchySF" returns CauchySF().
6. "FBSASF" returns FitnessBasedSelfAdaptiveSF().
7. "RGSF" returns RandomGaussianSF().

Usage

```
xegaDfScaleFactorFactory(method = "Const")
```

Arguments

method	A string specifying the scale factor function.

Details

In the literature, several approaches have been suggested. For a review see Sharma et al. (2019).

Value

A scale factor function for genes.

References

Sharma, Prashant; Sharma, Harish; Kumar, Sandeep; Bansal, Jagdish Chand (2019): A Review on Scale Factor Strategies in Differential Evolution Algorithm. pp. 925-934. In: Bansal, Jagdish Chand et al. (2019) Soft Computing for Problem Solving. Advances in Intelligent Systems and Computing, Vol. 817. Springer, Singapore, 2019. (ISBN:978-981-13-1594-7)

See Also

Other Configuration: `lFxegaDfGene`, `xegaDfCrossoverFactory()`, `xegaDfGeneMapFactory()`, `xegaDfMutationFactory()`, `xegaDfReplicationFactory()`

Examples

```
f<-xegaDfScaleFactorFactory("Const")
f(lFxegaDfGene)
f<-xegaDfScaleFactorFactory("Uniform")
f(lFxegaDfGene)
f(lFxegaDfGene)
```

---

xegaDfUCrossGene *Uniform crossover of 2 genes.*

Description

`xegaDfUCrossGene()` swaps alleles of both genes with a probability of 0.5. It generates a random mask which is used to build the new gene.

Usage

```
xegaDfUCrossGene(gg1, gg2, lF)
```

Arguments

gg1	Real-coded gene.
gg2	Real-coded gene.
lF	Local configuration of the genetic algorithm.

Value

Real-coded gene.

References

Syswerda, Gilbert (1989): Uniform Crossover in Genetic Algorithms. In: Schaffer, J. David (Ed.) Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, California, pp. 2-9. (ISBN:1-55860-066-3)

See Also

Other Crossover (Returns 1 Kid): `xegaDfCrossGene()`, `xegaDfUPCrossGene()`

Examples

```
gene1<-xegaDfInitGene(lFxegaDfGene)
gene2<-xegaDfInitGene(lFxegaDfGene)
gene3<-xegaDfUCrossGene(gene1, gene2, lFxegaDfGene)
```

---

xegaDfUPCrossGene *Parameterized uniform crossover of 2 genes.*

Description

`xegaDfUPCrossGene()` swaps alleles of both genes with a probability of `lF$UCrossSwap()`. It generate a random mask which is used to build the new gene.

Usage

```
xegaDfUPCrossGene(gg1, gg2, lF)
```

Arguments

`gg1`	Real-coded gene.
`gg2`	Real-coded gene.
`lF`	Local configuration of the genetic algorithm.

Value

Real-coded gene.

References

Spears William and De Jong, Kenneth (1991): On the Virtues of Parametrized Uniform Crossover. In: Belew, Richard K. and Booker, Lashon B. (Ed.) Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, California, pp. 230-236. (ISBN: 1-55860-208-9)

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Other Crossover (Returns 1 Kid): `xegaDfCrossGene()`, `xegaDfUCrossGene()`

Examples

```
gene1<-xegaDfInitGene(lFxegaDfGene)
gene2<-xegaDfInitGene(lFxegaDfGene)
gene3<-xegaDfUPCrossGene(gene1, gene2, lFxegaDfGene)
```

`xegaGedeInitGene` *Initialize a real-coded gene for grammar evolution*

Description

`xegaGedeInitGene()` generates a random real-coded gene with a given length.

Usage

```
xegaGedeInitGene(lF)
```

Arguments

`lF`	Local configuration of the genetic algorithm.

Details

In the real-coded representation of package `xegaDf`, *gene* is at least a list with

1. `$evaluated`: Boolean. TRUE if the fitness is known.
2. `$fit`: The fitness of the genotype of `$gene1`.
3. `$gene1`: a real vector (the genotype).

Value

A real-coded gene (a named list):

- `$evaluated`: FALSE. See package `xegaSelectGene`.
- `$evalFail`: FALSE. Set by the error handler(s) of the Evaluation Functions in package `xegaSelectGene` in the case of failure.
- `$fit`: Fitness.
- `$gene1`: A vector of reals.

References

Price, Kenneth V., Storn, Rainer M. and Lampinen, Jouni A. (2005) The Differential Evolution Algorithm (Chapter 2), pp. 37-134. In: Differential Evolution. A Practical Approach to Global Optimization. Springer, Berlin. <doi:10.1007/3-540-31306-0>

See Also

Other Intialization: `xegaDfInitGene()`

Examples

```
xegaGedeInitGene(lFxegaDfGene)
```

Index