Package ‘evola’

November 17, 2025

Version 1.0.7

Date 2025-12-01

Title Evolutionary Algorithm

Maintainer Giovanny Covarrubias-Pazaran <cova_ruber@live.com.mx>

Description Runs an evolutionary algorithm using the 'AlphaSimR' machinery <doi:10.1093/g3journal/jkaa017> .

Depends R(>= 3.5.0), AlphaSimR (>= 1.4.2), Matrix (>= 1.0), methods, crayon, enhancer

LazyLoad yes

License GPL (>= 2)

Author Giovanny Covarrubias-Pazaran [aut, cre] (ORCID: <https://orcid.org/0000-0002-7194-3837>)

Repository CRAN

Suggests rmarkdown, knitr

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

Date/Publication 2025-11-17 15:00:15 UTC

Contents

evola-package   **EVOL***utionary* **A***lgorithm*

Description

The evola package is nice wrapper of the AlphaSimR package that enables the use of the evolutionary algorithm to solve complex questions in a simple form.

The `evolafit` function is the core function of the package which allows the user to specify the problem and constraints to find a close-to-optimal solution using the evolutionary forces.

Keeping evola updated

The evola package is updated on CRAN every 4-months due to CRAN policies but you can find the latest source at https://github.com/covaruber/evola. This can be easily installed typing the following in the R console:

library(devtools)

install_github("covaruber/evola")

This is recommended if you reported a bug, was fixed and was immediately pushed to GitHub but not in CRAN until the next update.

Tutorials

For tutorials on how to perform different analysis with evola please look at the vignettes by typing in the terminal:

vignette("evola.intro")

Getting started

The package has been equiped with a couple of datasets to learn how to use the evola package:

* `DT_technow` dataset to perform optimal cross selection.

* `DT_wheat` dataset to perform optimal training population selection.

* `DT_cpdata` dataset to perform optimal individual.

Models Enabled

The machinery behind the scenes is AlphaSimR.

Bug report and contact

If you have any questions or suggestions please post it in https://stackoverflow.com or https://stats.stackexchange.com

I'll be glad to help or answer any question. I have spent a valuable amount of time developing this package. Please cite this package in your publication. Type 'citation("evola")' to know how to cite it.

Author(s)

Giovanny Covarrubias-Pazaran

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

Gaynor, R. Chris, Gregor Gorjanc, and John M. Hickey. 2021. AlphaSimR: an R package for breeding program simulations. G3 Gene|Genomes|Genetics 11(2):jkaa017. https://doi.org/10.1093/g3journal/jkaa017.

Chen GK, Marjoram P, Wall JD (2009). Fast and Flexible Simulation of DNA Sequence Data. Genome Research, 19, 136-142. http://genome.cshlp.org/content/19/1/136.

`addZeros` *Function to add zeros before and after a numeric vector to have the same number of characters.*

Description

Function to add zeros before and after a numeric vector to have the same number of characters.

Usage

```
addZeros(x, nr=2)
```

Arguments

`x`	Numeric vector.
`nr`	number of digits to keep to the right.

Details

A simple apply function to make a matrix of one row and nc columns.

Value

$res a matrix

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
addZeros(5)
```

---

bestSol *Extract the index of the best solution*

Description

Extracts the index of the best solution for all traits under the constraints specified.

Usage

```
bestSol(object, selectTop=TRUE, n=1)
```

Arguments

`object`	A resulting object from the function evolafit.
`selectTop`	Selects highest values for the fitness value if TRUE. Selects lowest values if FALSE.
`n`	An integer indicating how many solutions should be returned.

Details

A simple apply function looking at the fitness value of all the solution in the last generation to find the maximum value.

Value

$res the vector of best solutions in M for each trait in the problem

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
set.seed(1)
# Data
Gems <- data.frame(
  Color = c("Red", "Blue", "Purple", "Orange",
            "Green", "Pink", "White", "Black",
            "Yellow"),
  Weight = round(runif(9,0.5,5),2),
  Value = round(abs(rnorm(9,0,5))+0.5,2),
  Times=c(rep(1,8),0)
)
head(Gems)


# Task: Gem selection.
# Aim: Get highest combined value.
# Restriction: Max weight of the gem combined = 10.
res0<-evolafit(cbind(Weight,Value)~Color, dt= Gems,
               # constraints: if greater than this ignore
               constraintsUB = c(10,Inf),
               # constraints: if smaller than this ignore
               constraintsLB= c(-Inf,-Inf),
               # weight the traits for the selection
               b = c(0,1),
               # population parameters
               nCrosses = 100, nProgeny = 20, recombGens = 1,
               # coancestry parameters
               D=NULL, lambda=c(0,0), nQtlStart = 1,
               # selection parameters
               propSelBetween = .9, propSelWithin =0.9,
               nGenerations = 50
)

bestSol(res0$pop, n=2)
```

`drift` *Drift of positive allele calculation*

Description

This function takes a population and the simulation parameters from a .Pop class and calculates the current frequencies of the positive alleles for each trait.

Usage

```
drift(pop, simParam, solution=NULL, traits=1)
```

Arguments

`pop`	an object of class "Pop"
`simParam`	the simulation parameters stored in the evolaMod object.
`solution`	an alternative RRBLUP solution so the frequencies of positive alleles are calculated for a given SNP chip. If NULL the frequencies are calculated for the real QTLs.
`traits`	traits considered.

Details

A simple apply function to get all the QTLs, segregation sites, positive alleles and frecuency of the positive allele for each trait in a population.

Value

$res a matrix

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
#Create founder haplotypes
founderPop = quickHaplo(nInd=10, nChr=1, segSites=30)

#Set simulation parameters
SP = SimParam$new(founderPop)

SP$addTraitA(10)
SP$setVarE(h2=0.5)
SP$addSnpChip(15)

#Create population
pop = newPop(founderPop, simParam=SP)

# drift for QTLs
drift(pop, simParam = SP )



ans = RRBLUP(pop, simParam=SP)

# drift for average allelic effects
```

```
drift(pop, simParam = SP, solution = ans)
```

---

evolafit *Fits a genetic algorithm for a set of traits and constraints.*

Description

Using the AlphaSimR machinery it recreates the evolutionary forces applied to a problem where possible solutions replace individuals and combinations of variables to optimize in the problem replace the genes or QTLs. Then evolutionary forces (mutation, selection and drift) are applied to find a close-to-optimal solution. Although multiple traits are enabled it is assumed that same QTLs are behind all the traits, differing only in their average allelic effects.

Usage

```
evolafit(formula, dt,
     constraintsUB, constraintsLB, constraintW=NULL,
     b, nCrosses=50, nProgeny=20,nGenerations=20,
     recombGens=1, nChr=1, mutRateAllele=0,  mutRateAlpha=0,
     nQtlStart=NULL, D=NULL, lambda=0,
     propSelBetween=NULL,propSelWithin=NULL,
     fitnessf=NULL, verbose=TRUE, dateWarning=TRUE,
     selectTop=TRUE, tolVarG=1e-6,
     Ne=50, initPop=NULL, simParam=NULL,
     fixNumQtlPerInd=FALSE, traceDelta=TRUE, topN=10,
     includeSet=NULL, excludeSet=NULL, haplo=NULL,
     ...)
```

Arguments

`formula`	Formula of the form *y~x where* `y` refers to the average allelic substitution effects of the QTLs (alpha) for each trait, and `x` refers to the variable defining the genes or QTLs to be combined in the possible solutions.
`dt`	A dataset containing the average allelic effects (a) and classifiers/genes/QTLs.
`constraintsUB`	A numeric vector specifying the upper bound constraints for the breeding values applied at each trait. The length is equal to the number of traits/features in the formula. If missing is assume an infinite value for all traits. Solutions (individuals in the population) with higher value than the upper bound are assigned a -infinite value if the argument `selectTop=TRUE` and to +infinite when `selectTop=FALSE`, which is equivalent to reject/discard a solution based on the fitness function.

`constraintsLB`	A numeric vector specifying the lower bound constraints for the breeding values applied at each trait. The length is equal to the number of traits/features in the formula. If missing is assume a -infinite value for all traits. Solutions with lower value than the lower bound are assigned a +infinite value if the argument `selectTop=TRUE` and to -infinite when `selectTop=FALSE`, which is equivalent to reject/discard a solution based on the fitness function.
`constraintW`	A numeric vector of length equal to the number of generations to specify the weights to be applied to the lower and upper bound constraint values to relax the constraints if needed. If values is NULL a vector of 1s with length equal to the number of generations is created.
`b`	A numeric vector specifying the weights that each trait has in the fitness function (i.e., a selection index). The length should be equal to the number of traits/features. If missing is assumed equal weight (1) for all traits.
`nCrosses`	A numeric value indicating how many crosses should occur in the population of solutions at every generation.
`nProgeny`	A numeric value indicating how many progeny (solutions) each cross should generate in the population of solutions at every generation.
`nGenerations`	The number of generations that the evolutionary process should run for.
`recombGens`	The number of recombination generations that should occur before selection is applied. This is in case the user wants to allow for more recombination before selection operates. The default is 1.
`nChr`	The number of chromosomes where features/genes should be allocated to. The default value is 1 but this number can be increased to mimic more recombination events at every generation and avoid linkage disequilibrium.
`mutRateAllele`	A value between 0 and 1 to indicate the proportion of random QTL dosage that should mutate in each individual. For example, a value of 0.1 means that a random 10% of the QTLs will mutate in each individual randomly taking values of 0 or 1. Is important to notice that this implies that the stopping criteria based in variance will never be reached because we keep introducing variance through random mutation.
`mutRateAlpha`	A value between 0 and 1 to indicate the proportion of random QTL average allelic effects that should mutate in each individual. For example, a value of 0.1 means that a random 10% of the QTLs will mutate in each individual randomly taking values of 0 or 1. Is important to notice that this implies that the stopping criteria based in variance will never be reached because we keep introducing variance through random mutation.
`nQtlStart`	The number of QTLs/genes (classifier `x` in the formula) that should be fixed for the positive allele at the begginning of the simulation. If not specified it will be equal to the 20% of the QTLs (calculated as the number of rows in the dt argument over 5). This is just an initial value and will change as the population evolve under the constraints specified by the user. See details section.
`D`	A relationship matrix between the QTLs (a kind of linkage disequilibrium) specified in the right side of the formula (levels of the `x` variable). This matrix can be used or ignored in the fitness function. By default the weight to the q'Dq component is 0 though the lambda argument, where x is an individual in the population of a solution.

`lambda`	A numeric value indicating the weight assigned to the relationship between QTLs in the fitness function. If not specified is assumed to be 0. This can be used or ignored in your customized fitness function.
`propSelBetween`	A numeric value between 0 and 1 indicating the proportion of families/crosses of solutions/individuals that should be selected. The default is 1, meaning all crosses are selected or passed to the next generation.
`propSelWithin`	A numeric value between 0 and 1 indicating the proportion of individuals/solutions within families/crosses that should be selected. The default value is 0.5, meaning that 50% of the top individuals are selected.
`fitnessf`	An alternative fitness function to be applied at the level of individuals or solutions. It could be a linear combination of the trait breeding values. The available variables internally are:

Y: matrix of trait breeding values for the individuals/solutions. Of dimensions s x t, s soultions and t traits.

b: vector of trait weights, specified in the 'b' argument. Of dimensions t x 1, t traits by 1

Q: matrix with QTLs for the individuals/solutions. Of dimensions s x p, s solutions and p QTL columns. Although multiple traits are enabled it is assumed that same QTLs are behind all the traits, differing only in their average allelic effects.

D: matrix of relationship between the QTLs, specified in the 'D' argument. Of dimensions p x p, for p QTL columns

lambda: a numeric value indicating the weight assigned to the relationship between QTLs in the fitness function. If not specified is assumed to be 0. This can be used or ignored in your customized fitness function.

a: list of vectors with average allelic effects for a given trait. Of dimensions s x 1, s solutions by 1 column

If `fitnessf=NULL`, the default function will be the `ocsFun` function:

```
function(Y,b,d,Q,D,a,lambda){(Y%*%b) - d}
```

where `(Y%*%b)` is equivalent to [(Q'a)b] in genetic contribution theory, and `d` is equal to the diagonal values from Q'DQ from contribution theory,

If you provide your own fitness function please keep in mind that the variables Y, b, Q, D, a, and lambda are already reserved and these variables should always be added to your function (even if you do not use them) in addition to your new variables so the machinery runs.

An additional fitness function for accounting only for the group relationship is `inbFun` when the user wants to find solutions that maximize the representativeness of a sample and the `D` argument is not NULL. You will need to select the solutions with lower values (`selectTop=TRUE`) which indicate solutions with more representativeness and you may need to indicate lower bound constraints (`constraintsLB`).

An additional fitness function available for regression problems is `regFun` but is not the default since it would require additional arguments not available in a regular genetic algorithm problem (e.g., `y` and `X` to compute `y-Xb`).

`verbose`	A logical value indicating if we should print logs.

`dateWarning`	A logical value indicating if you should be warned when there is a new version on `CRAN`.
`selectTop`	Selects highest values for the fitness value if `TRUE`. Selects lowest values if `FALSE`.
`tolVarG`	A stopping criteria (tolerance for genetic variance) when the variance across traits is smaller than this value, which is equivalent to assume that all solutions having the same QTL profile (depleted variance). The default value is `1e-6` and is computed as the sum of the diagonal values of the genetic variance covariance matrix between traits.
`Ne`	initial number of founders in the population (will be important for long term sustainability of genetic variance).
`initPop`	an object of `Pop-class`.
`simParam`	an object of `SimParam`.
`fixNumQtlPerInd`	A `TRUE/FALSE` value to indicate if we should fix the argument nQtlStart across all generations. This should be used with care since this is not how usually genetic algorithms work and in my experience only using GA for regression problems is a special case where this argument should be set to TRUE. The behavior assumes that if set to TRUE and a particular solution has more QTLs active than nQtlStart some QTLs will be set to 0 and if a solution has less QTLs active than nQtlStart some QTLs will be activated. All activations or deactivations are done at random. This could be an alternative to use a counting trait to restrain the number of QTLs active in a solution but is slower.
`traceDelta`	a logical value indicating if we should compute the rate of coancestry Q'DQ at each iteration. This metric is used by the pareto plot but is not needed for the evolutionary process and it can take a considerable amount of time when the number of QTLs is big.
`topN`	an integer value indicating the maximum number of solutions to keep in each generation.
`includeSet`	a numeric vector with 0s and 1s of length equal to the number of QTLs (number of rows in your dataset) to indicate which individuals (1s) should be forced to be activated.
`excludeSet`	a numeric vector with 0s and 1s of length equal to the number of QTLs (number of rows in your dataset) to indicate which individuals (1s) should be forced to be deactivated.
`haplo`	an alternative way to provide the initial solutions (founders) to manage the starting point or activated QTLs. See the vignettes for examples. In general, the user provides a matrix of raw calls (8-bit) with dimensions n rows (founders) and p columns (QTLs) with 0s and 1s to control the initial founders which will also be considered in the first generation of selection. If NULL (default) the program will randomly activate n QTLs according to the `nQtlStart` argument.
`...`	Further arguments to be passed to the fitness function if required.

Details

Using the `AlphaSimR` machinery (runMacs) it recreates the evolutionary forces applied to a problem where possible solutions replace individuals and combinations of variables in the problem replace the genes. Then evolutionary forces are applied to find a close-to-optimal solution. The number of solutions are controlled with the nCrosses and nProgeny parameters, whereas the number of initial QTLs activated in a solution is controlled by the nQtlStart parameter. The number of activated QTLs of course will increase if has a positive effect in the fitness of the solutions. The drift force can be controlled by the recombGens parameter. The mutation rate can be controlled with the mutRateAllele parameter. The recombination rate can be controlled with the nChr argument.

The `indivPerformance` output slot contains the columns id, fitness, generation, nQTLs, and deltaC. These mean the following:

In fitness : represents the fitness function value of a solution.

In deltaC : it represents the change in coancestry (e.g., inbreeding), it can be thought as the rate of coancestry. It is calculated as q'Dq where 'q' represents the contribution vector, 'D' is the linkage disequilibrium matrix between QTNs (whatever the QTNs represent for your specific problem). In practice we do QAQ' and extract the diagonal values.

In generation : it represents the generation at which this solution appeared.

In nQTNs : it represent the final number of QTNs that are activated in homozygote state for the positive effect.

During the run the columns printed in the console mean the following:

generation: generation of reproduction

constrainedUB: number of solutions constrained by the upper bound specified

constrainedLB: number of solutions constrained by the lower bound specified

varG: genetic variance present in the population due to the QTNs

propB: proportion of families selected during that iteration

propW: proportion of individuals within a family selected in that iteration

time: the time when the iteration has finished.

Value

indivPerformance the matrix of fitness, deltaC, generation, nQTNs per solution per generation. See details section above.

pedBest contains the pedigree of the selected solutions across iterations.

$score a matrix with scores for different metrics across n generations of evolution.

$pheno the matrix of phenotypes of individuals/solutions present in the last generation.

pop AlphaSimR object used for the evolutionary algorithm at the last iteration.

constCheckUB A matrix with as many rows as solutions and columns as traits to be constrained. 0s indicate that such trait went beyond the bound in that particular solution.

constCheckLB A matrix with as many rows as solutions and columns as traits to be constrained. 0s indicate that such trait went beyond the bound in that particular solution.

traits a character vector corresponding to the name of the variables used in the fitness function.

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

Gaynor, R. Chris, Gregor Gorjanc, and John M. Hickey. 2021. AlphaSimR: an R package for breeding program simulations. G3 Gene|Genomes|Genetics 11(2):jkaa017. https://doi.org/10.1093/g3journal/jkaa017.

Chen GK, Marjoram P, Wall JD (2009). Fast and Flexible Simulation of DNA Sequence Data. Genome Research, 19, 136-142. http://genome.cshlp.org/content/19/1/136.

See Also

`evolafit` – the information of the package

Examples

```
set.seed(1)

# Data
Gems <- data.frame(
  Color = c("Red", "Blue", "Purple", "Orange",
            "Green", "Pink", "White", "Black",
            "Yellow"),
  Weight = round(runif(9,0.5,5),2),
  Value = round(abs(rnorm(9,0,5))+0.5,2),
  Times=c(rep(1,8),0)
)
head(Gems)
#     Color Weight Value
# 1     Red   4.88  9.95
# 2    Blue   1.43  2.73
# 3  Purple   1.52  2.60
# 4  Orange   3.11  0.61
# 5   Green   2.49  0.77
# 6    Pink   3.53  1.99
# 7   White   0.62  9.64
# 8   Black   2.59  1.14
# 9  Yellow   1.77 10.21



# Task: Gem selection.
# Aim: Get highest combined value.
# Restriction: Max weight of the gem combined = 10.

# simple specification
res00<-evolafit(formula=cbind(Weight,Value)~Color, dt= Gems,
               # constraints on traits: if greater than this ignore
               constraintsUB = c(10,Inf), nGenerations = 10
)
best = bestSol(res00$pop)[,"fitness"]
Q <- pullQtlGeno(res00$pop, simParam = res00$simParam, trait=1); Q <- Q/2
qa = Q[best,] %*% as.matrix(Gems[,c("Weight","Value")]); qa
```

```
# more complete specification
res0<-evolafit(formula=cbind(Weight,Value)~Color, dt= Gems,
               # constraints on traits: if greater than this ignore
               constraintsUB = c(10,Inf),
               # constraints on traits: if smaller than this ignore
               constraintsLB= c(-Inf,-Inf),
               # weight the traits for the selection (fitness function)
               b = c(0,1),
               # population parameters
               nCrosses = 100, nProgeny = 20,
               # genome parameters
               recombGens = 1, nChr=1,  mutRateAllele=0, nQtlStart = 2,
               # coancestry parameters
               D=NULL, lambda=0,
               # selection parameters
               propSelBetween = .9, propSelWithin =0.9,
               nGenerations = 50
)

Q <- pullQtlGeno(res0$pop, simParam = res0$simParam, trait=2); Q <- Q/2
best = bestSol(res0$pop)[,"fitness"]
qa = Q[best,] %*% as.matrix(Gems[,c("Weight","Value")]); qa
Q[best,]

# $`Genes`
# Red   Blue Purple Orange  Green   Pink  White  Black Yellow
# 1      1      0      0      1      0      0      1      0
#
# $Result
# Weight  Value
# 8.74  32.10
evolmonitor(res0)
pareto(res0)
```

---

evolaPop-class *Genetic algorithm pop*

Description

A genetic algorithm pop fit by evolafit. This class extends class "Pop" class and includes some additional slots.

Objects from the Class

Objects are created by calls to the evolafit function.

Slots

indivPerformance the matrix of q'a (score), deltaC, q'Dq, generation, nQTNs per solution per generation. See details section above. All other slots are inherited from class "Pop".

pedBest if the argument keepBest=TRUE this contains the pedigree of the selected solutions across iterations. All other slots are inherited from class "Pop".

$score a matrix with scores for different metrics across n generations of evolution. All other slots are inherited from class "Pop".

$pheno the matrix of phenotypes of individuals/solutions present in the last generation. All other slots are inherited from class "Pop".

$phenoBest the matrix of phenotypes of top (parents) individuals/solutions present in the last generation. All other slots are inherited from class "Pop".

constCheckUB A matrix with as many rows as solutions and columns as traits to be constrained. 0s indicate that such trait went beyond the bound in that particular solution. All other slots are inherited from class "Pop".

constCheckLB A matrix with as many rows as solutions and columns as traits to be constrained. 0s indicate that such trait went beyond the bound in that particular solution. All other slots are inherited from class "Pop".

traits a character vector corresponding to the name of the variables used in the fitness function. All other slots are inherited from class "Pop".

Extends

Class "Pop", directly.

Methods

update `signature(object = "evolaPop")`: also a non-method for the same reason as `update`

See Also

`evolafit`

Examples

```
showClass("evolaPop")
```

`evolmonitor` *plot the change of values across iterations*

Description

`plot` for monitoring.

Usage

```
evolmonitor(object, kind, ...)
```

Arguments

`object`	model object of class `"evolafit"`
`kind`	a numeric value indicating what to plot according to the following values: 1: Average and best q'a (contribution) 2. Average q'Dq and deltaC 3. Number of QTLs activated
`...`	Further arguments to be passed to the plot function.

Value

trace plot

Author(s)

Giovanny Covarrubias

See Also

`plot`, `evolafit`

`freqPosAllele` *Extract frequency of positive alleles*

Description

Computes the frequency of positive alleles given certain marker effects.

Usage

```
freqPosAllele(M, alpha)
```

Arguments

`M`	Marker matrix.
`alpha`	Vector of marker effects.

Details

A simple apply function to compute the frequency of the positive alleles.

Value

$res a matrix

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
#random population of 10 lines with 5 markers
M <- matrix(rep(0,10*5),10,5)
for (i in 1:10) {
  M[i,] <- ifelse(runif(5)<0.5,0,2)
}
alpha <- sample(c(-1,1),5, replace=TRUE)

freqPosAllele(M, alpha)
```

`importHaploSparse` *Import haplotypes*

Description

Formats haplotype in a matrix format to an AlphaSimR population that can be used to initialize a simulation. This function serves as wrapper for newMapPop that utilizes a more user friendly input format.

Usage

```
importHaploSparse(haplo, genMap, ploidy = 2L, ped = NULL)
```

Arguments

`haplo`	a sparse matrix of haplotypes
`genMap`	genetic map as a data.frame. The first three columns must be: marker name, chromosome, and map position (Morgans). Marker name and chromosome are coerced using as.character. See importGenMap.
`ploidy`	ploidy level of the organism.
`ped`	an optional pedigree for the supplied genotypes. See details.

Details

The optional pedigree can be a data.frame, matrix or a vector. If the object is a data.frame or matrix, the first three columns must include information in the following order: id, mother, and father. All values are coerced using as.character. If the object is a vector, it is assumed to only include the id. In this case, the mother and father will be set to "0" for all individuals.

Value

$res a MapPop-class if ped is NULL, otherwise a NamedMapPop-class

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
haplo <- Matrix::Matrix(0, nrow=4, ncol=5)
for (i in 1:4) {
  haplo[i,] <- ifelse(runif(5)<0.2,0,1)
}
colnames(haplo) = letters[1:5]

genMap = data.frame(markerName=letters[1:5],
                    chromosome=c(1,1,1,2,2),
                    position=c(0,0.5,1,0.15,0.4))

ped = data.frame(id=c("a","b"),
                 mother=c(0,0),
                 father=c(0,0))

founderPop = importHaploSparse(haplo=haplo,
                         genMap=genMap,
                         ploidy=2L,
                         ped=ped)
```

`inbFun` *Fitness function from contribution theory using only the group relationship*

Description

Simple function for fitness where we only use the group relationship.

Usage

```
inbFun(Q,D,...)
```

Arguments

`Q`	A QTL matrix. See details.
`D`	An LD matrix. See details.
`...`	additional arguments to pass.

Details

A simple apply function of a regular index weighted by a vector of relationships.

Matrix::diag(Q%*%Matrix::tcrossprod(D,Q)) of dimensions n x n

Notice that Q represents the marker of QTLs (columns) for all solutions (rows) and D the LD between QTLs. The user can modify this function as needed and provide it to the evolafit function along with other arguments.

Value

$res a vector of values

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
Q <- matrix(1,3,3) # QTL matrix available internally
D <- diag(3) # LD matrix
inbFun(Q=Q, D=D) # group relationship
```

`Jc` *Matrix of ones*

Description

Makes a matrix of ones with a single row and nc columns.

Usage

```
Jc(nc)
```

Arguments

`nc`	Number of columns to create.

Details

A simple apply function to make a matrix of one row and nc columns.

Value

$res a matrix

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
Jc(5)
```

`Jr` *Matrix of ones*

Description

Makes a matrix of ones with a single column and nr rows.

Usage

```
Jr(nr)
```

Arguments

`nr`	Number of rows to create.

Details

A simple apply function to make a matrix of one column and nr rows.

Value

$res a matrix

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

evolafit – the core function of the package

Examples

```
Jr(5)
```

---

nQtl *Matrix of number of activated QTLs*

Description

Makes a matrix indicating how many QTLs were activated for each solution.

Usage

```
nQtl(object)
```

Arguments

object	Object returned by the evolafit function.

Details

A simple apply function to count the number of active QTLs per solution (row) per trait (columns).

Value

$res a matrix

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

evolafit – the core function of the package

Examples

```
set.seed(1)

# Data
Gems <- data.frame(
  Color = c("Red", "Blue", "Purple", "Orange",
            "Green", "Pink", "White", "Black",
            "Yellow"),
  Weight = round(runif(9,0.5,5),2),
  Value = round(abs(rnorm(9,0,5))+0.5,2),
  Times=c(rep(1,8),0)
)
head(Gems)



# Task: Gem selection.
# Aim: Get highest combined value.
# Restriction: Max weight of the gem combined = 10.

# simple specification
res00<-evolafit(formula=cbind(Weight,Value)~Color, dt= Gems,
               # constraints on traits: if greater than this ignore
               constraintsUB = c(10,Inf), nGenerations = 10
)
nQtl(res00)
```

`ocsFun` *Fitness function from contribution theory*

Description

Simple function for fitness where an index of traits is weighted by the group relationship.

Usage

```
ocsFun(Y,b,Q,D,lambda,scaled=TRUE,...)
```

Arguments

Y	A matrix of trait values. See details.
b	A vector of trait weights. See details.
Q	A QTL matrix for solutions. See details.
D	An LD matrix for solutions. See details.

`lambda`	A numeric value to weight the Q'DQ portion of the objective function (to be provided by the user with the lambda argument). See details.
`scaled`	A logical value to indicate if traits should be scaled prior to multiply by the weights.
`...`	additional arguments to pass.

Details

A simple apply function of a regular index weighted by a vector of relationships.

Y%*%b - d

Internally, we use this function in the following way:

The Y matrix is the matrix of trait-GEBVs (Y = Q%*%alpha) for each solution, and b is the user-specified trait weights.

d = qtDq * lambda; where qtDq is equal to Matrix::diag(Q%*%Matrix::tcrossprod(D,Q)) of dimensions n x n

Notice that Q represents the marker of QTLs (columns) for all solutions (rows) and D the LD between QTLs. The user can modify this function as needed and provide it to the evolafit function along with other arguments.

Value

$res a vector of values

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
Y=matrix(1:12,4,3) # 4 solutions with 3 traits

Q=matrix(sample(0:1,32,replace = TRUE),nrow=4,ncol=8) # coancestry for each solution

D=diag(8)

b=rep(1,3)

lambda=0.5

ocsFun(Y=Y,Q=Q,D=D,b=b, lambda=lambda) # Yb - d where d is QAQ' and A is the LD between QTNs
```

`ocsFunC` *Fitness function from contribution theory*

Description

Simple function for fitness where an index of traits is weighted by the group relationship.

Usage

```
ocsFunC(Q,
        SNP, solution,
        wtf=NULL,
        wbaf="base",
        ...)
```

Arguments

`Q`	A QTL matrix for solutions. See details.
`SNP`	A SNP marker matrix for the QTLs if the QTLs are individuals.
`solution`	an object of RRsol-class.
`wtf`	A vector of trait weights for the solution object. See details.
`wbaf`	A character value to indicate which type of weights should be used for the beneficial-allele frequencies of the different solutions (Q). Accepted values are; 1) 'base' to use 1-benef.freq.base.pop as proposed originaly by Covarrubias-Pazaran and Pita (2026), 2) 'alpha' to use the average allelic effects, 3) 'none' to only multuply by one which is equivalent to not using weights.
`...`	additional arguments to pass.

Details

A function of the type:

sum(freq.solution * (1 - freq.base.pop))

the frequency of beneficial alleles is computed for each solution in Q and then it is weighted by 1 - freq.base.pop which is the frequency of beneficial alleles in the base population.

Value

$res a vector of values

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

evolafit – the core function of the package

Examples

```
# mickey mouse example

Q=matrix(sample(0:1,32,replace = TRUE),nrow=4,ncol=8) # possible solutions

SNP=matrix(sample(0:2,80,replace = TRUE),nrow=8,ncol=10) # SNPs of 8 individuals with 10 markers

solution = matrix(rnorm(10)) # marker effects

ocsFunC(SNP=SNP,Q=Q, solution=solution) # sum of frequency of positive alleles

 ## Not run:

  # using the function in an optimal contribution

  data(DT_technow, package="enhancer")
  DT <- DT_technow
  DT$occ <- 1; DT$occ[1]=0

  Md <- apply(Md_technow,2,as.numeric)
  rownames(Md) <- rownames(Md_technow)
  Mf <- apply(Mf_technow,2,as.numeric)
  rownames(Mf) <- rownames(Mf_technow)
  M <- rbind(Md,Mf)

  A <- A.matr(M)
  A <- A[DT$hy,DT$hy]

  library(rrBLUP)
  mixm = mixed.solve(y=DT$GY, Z=M)
  solution = matrix(mixm$u[colnames(M)])

  sum(freqPosAllele(M,solution[,1])) # starting frequencies

  # run the genetic algorithm
  # we assig a weight to x'Dx of (20*pi)/180=0.34
  res<-evolafit(formula = c(GY, occ)~hy,
              dt= DT,
              # constraints: if sum is greater than this ignore
              constraintsUB = c(Inf,100),
              # constraints: if sum is smaller than this ignore
              constraintsLB= c(-Inf,-Inf),
              # weight the traits for the selection
              b = c(1,0),
              # population parameters
              nCrosses = 100, nProgeny = 10,
              recombGens=1, nChr=1, mutRateAllele=0,
              # coancestry parameters
              D=A, lambda= (20*pi)/180 , nQtlStart = 90,
              # selection parameters
              propSelBetween = 0.5, propSelWithin =0.5,
              fitnessf = ocsFunC,
              SNP=M, solution=solution,
              nGenerations = 20)

  Q <- pullQtlGeno(res$pop, simParam = res$simParam, trait=1); Q <- Q/2
  best = bestSol(res$pop)[,"fitness"]
  qa = (Q %*% DT$GY)[best,]; qa
  qAq = Q[best,] %*% A %*% Q[best,]; qAq
  sum(Q[best,]) # total # of inds selected

  sum(freqPosAllele(diag(Q[best,])%*%M,solution[,1]))

  evolmonitor(res)
  plot(DT$GY, col=as.factor(Q[best,]),
     pch=(Q[best,]*19)+1)

  pareto(res)


## End(Not run)
```

---

pareto *plot the change of values across iterations*

Description

`plot` for monitoring.

Usage

```
pareto(object, scaled=TRUE,pch=20, xlim, ...)
```

Arguments

`object`	model object returned by `"evolafit"`
`scaled`	a logical value to specify the scale of the y-axis (gain in merit).
`pch`	symbol for plotting points as desribed in par
`xlim`	upper and lower bound in the x-axis
`...`	Further arguments to be passed to the plot function.

Value

vector of plot

Author(s)

Giovanny Covarrubias

See Also

`plot`, `evolafit`

`regFun` *Fitness function from linear regressions based on mean squared error.*

Description

Simple function for fitness where the mean squared error is computed when the user provides y and X and b are the average allelic effects of the population in the genetic algorithm.

Usage

```
regFun(Q,a,X,y,...)
```

Arguments

`Q`	A QTL matrix for the proposed solutions (internal matrix). See details.
`a`	A named list with vectors of average allelic effects per trait (internal matrix). See details.
`X`	A matrix of covariates or explanatory variables (to be provided by the user in the ... arguments). See details.
`y`	A vector of the response variable (to be provided by the user in the ... arguments). See details.
`...`	additional arguments to pass.

Details

A simple apply function of a regular mean squared error.

`( y - X%*%b ) ^ 2`

Internally, we use this function in the following way:

The y vector and X matrix are provided by the user and are fixed values that do not change across iterations. The evolutionary algorithm optimizes the b values which are the QTLs and associated average allelic effects that are evolving. The 'b' coefficients in the formula come from the GA and are computed as:

`b[j] = a[[1]][p[[j]]]`

where a[[1]] is the list of QTL average allelic effects per trait provided in the original dataset, whereas p is a list (with length equal to the number of solutions) that indicates which QTLs are activated in each solution and the j variable is just a counter so each solution is tested.

Value

$res a vector of values

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

`evolafit` – the core function of the package

Examples

```
y <- rnorm(40) # 4 responses
X=matrix(rnorm(120),40,3) # covariates
Q=matrix(0,40,30) # QTL matrix with 30 QTLs
for(i in 1:nrow(Q)){Q[i,sample(1:ncol(Q),3)]=1}
a <- matrix(rnorm(30),ncol=1) # 30 average allelic effects in trait 1

mse = regFun(y=y, X=X, Q=Q, a=a, # used
             # ignored, Y is normally available in the evolafit routine
             Y=X)
```

`summary.Pop` *summary form an evolafit model*

Description

`summary` method for class `"Pop"`.

Usage

```
## S3 method for class 'Pop'
summary(object, ...)
```

Arguments

`object`	an object of class `"Pop"`
`...`	Further arguments to be passed

Value

an updated model

Author(s)

Giovanny Covarrubias

Examples

```
set.seed(1)

# Data
Gems <- data.frame(
  Color = c("Red", "Blue", "Purple", "Orange",
            "Green", "Pink", "White", "Black",
            "Yellow"),
  Weight = round(runif(9,0.5,5),2),
  Value = round(abs(rnorm(9,0,5))+0.5,2),
  Times=c(rep(1,8),0)
)
# Task: Gem selection.
# Aim: Get highest combined value.
# Restriction: Max weight of the gem combined = 10.

# simple specification
res<-evolafit(formula=cbind(Weight,Value)~Color, dt= Gems,
               # constraints on traits: if greater than this ignore
               constraintsUB = c(10,Inf), nGenerations = 2
)
summary(res$pop)
```

update.evolaFitMod *update form an evolafit model*

Description

update method for class "evolaFitMod".

Usage

```
## S3 method for class 'evolaFitMod'
update(object, formula., evaluate = TRUE, ...)
```

Arguments

object	an object of class "evolaFitMod"
formula.	an optional formula
evaluate	a logical value to indicate if an evaluation of the call should be done or not
...	Further arguments to be passed

Value

an updated model

Author(s)

Giovanny Covarrubias

Examples

```
set.seed(1)

# Data
Gems <- data.frame(
  Color = c("Red", "Blue", "Purple", "Orange",
            "Green", "Pink", "White", "Black",
            "Yellow"),
  Weight = round(runif(9,0.5,5),2),
  Value = round(abs(rnorm(9,0,5))+0.5,2),
  Times=c(rep(1,8),0)
)
# Task: Gem selection.
# Aim: Get highest combined value.
# Restriction: Max weight of the gem combined = 10.

# simple specification
res<-evolafit(formula=cbind(Weight,Value)~Color, dt= Gems,
               # constraints on traits: if greater than this ignore
               constraintsUB = c(10,Inf), nGenerations = 2
)
resUp=update(res)
```

---

varQ *Extract the variance existing in the genome solutions*

Description

Extracts the variance found across the M element of the resulting object of the evolafit() function which contains the different solution and somehow represents the genome of the population.

Usage

```
varQ(object)
```

Arguments

`object`	A resulting object from the function evolafit.

Details

A simple apply function looking at the variance in each column of the M element of the resulting object of the evolafit function.

Value

$res a value of variance

References

Giovanny Covarrubias-Pazaran (2024). evola: a simple evolutionary algorithm for complex problems. To be submitted to Bioinformatics.

See Also

evolafit – the core function of the package

Examples

```
set.seed(1)
# Data
Gems <- data.frame(
  Color = c("Red", "Blue", "Purple", "Orange",
            "Green", "Pink", "White", "Black",
            "Yellow"),
  Weight = round(runif(9,0.5,5),2),
  Value = round(abs(rnorm(9,0,5))+0.5,2),
  Times=c(rep(1,8),0)
)
head(Gems)


# Task: Gem selection.
# Aim: Get highest combined value.
# Restriction: Max weight of the gem combined = 10.
res0<-evolafit(cbind(Weight,Value)~Color, dt= Gems,
               # constraints: if greater than this ignore
               constraintsUB = c(10,Inf),
               # constraints: if smaller than this ignore
               constraintsLB= c(-Inf,-Inf),
               # weight the traits for the selection
               b = c(0,1),
               # population parameters
               nCrosses = 100, nProgeny = 20, recombGens = 1,
               # coancestry parameters
               D=NULL, lambda=c(0,0), nQtlStart = 1,
               # selection parameters
               propSelBetween = .9, propSelWithin =0.9,
               nGenerations = 5
)

varQ(res0)
```

Index