

# Package ‘glads’

March 30, 2024

**Type** Package

**Title** Genomic Landscape of Divergence Simulation

**Version** 0.1.8

**Author** Claudio S. Quilodran, Kristen Ruegg, Ashley T. Sendell-Price, Sonya Clegg, Tim Coulson, Eric Anderson.

**Maintainer** Claudio S. Quilodrán <claudio.quilodran@unige.ch>

**Description** The glads package is an individual-based framework for forward demographic, genetic and genomic simulations. The main aim of the R package is therefore to simulate the divergence of populations further in time, elucidating genomic patterns that may be generated by a range of demographic and genetic processes.

**License** GPL-2

**LazyData** TRUE

**Imports** dplyr,  
magrittr,  
pegas,  
progress,  
Rcpp (>= 0.12.7)

**LinkingTo** Rcpp

**RoxygenNote** 7.2.3

**Encoding** UTF-8

## R topics documented:

glads-package . . . . .	2
evolve . . . . .	2
fitness . . . . .	13
initial.struct . . . . .	14
phenotype . . . . .	15
struct2pegas . . . . .	17
<b>Index</b>	<b>18</b>

---

 glads-package

*Genomic Landscape of Divergence Simulation*


---

### Description

The package glads is an individual-based framework for forward demographic, genetic and genomic simulations. The main aim of the R package is to simulate the divergence of populations further in time, elucidating genomic patterns that may be generated by a range of demographic and genetic processes.

### Details

The list of functions can be displayed with `library(help = glads)`.

The main function of the package glads is "evolve", which allows the forward simulation of the genetic structure of populations under various scenarios of drift and selection. The supported data are allelic frequencies, SNPs or sequences of DNA. User may either simulate a theoretical dataset or directly work with their own empirical observations. In any case, the observation should be arranged in a two-dimensional array that represents a pair of homologous chromosomes. Each element of the array is an integer defining the copy of a given allele at a given locus.

---

 evolve

*Evolution of genetic and genomic landscapes*


---

### Description

This function simulates the evolution of individual-based populations forward in time.

### Usage

```
evolve(
  x,
  time,
  type = c("constant", "dynamic", "additive", "custom"),
  recombination = c("map", "average"),
  recom.rate,
  loci.pos = NULL,
  chromo_mb = NULL,
  init.sex = NULL,
  migration.rate = NULL,
  mutation.rate = NULL,
  param.z = NULL,
  param.w = NULL,
  fun = c(phenotype = NULL, fitness = NULL),
  admixture = list(taxa = NULL, dd = FALSE, eqvm = FALSE, nmin = 10)
)
```

**Arguments**

<code>x</code>	List of objects of class "struct" with the initial genetic structure of populations.
<code>time</code>	Number of simulated generations. It could also be a two-element vector with the number of simulated generations and the elapsed time to record the outputs.
<code>type</code>	Type of simulated evolution. Current options are 'constant', 'dynamic', 'additive', and 'custom' (see Details).
<code>recombination</code>	Type of recombination between homologous chromosomes. Current options are 'map' and 'average' (see Details).
<code>recom.rate</code>	A numerical value for the recombination type 'average', or a vector of $nl - 1$ elements, with the recombination rate between neighbouring loci for type 'map' ( $nl$ : number of loci) (see Details).
<code>loci.pos</code>	A vector with the position of loci. The default value is NULL, but it is required for the recombination type 'average'.
<code>chromo_mb</code>	A numerical value with the size of the simulated chromosome (in megabase). The default value is NULL, but it is required for the recombination type 'average'.
<code>init.sex</code>	A list of vectors defining the sex of the initial individuals in the populations (1: females and 2: males). The default value is NULL and assigns the sex of the first generation randomly.
<code>migration.rate</code>	A single value setting the migration rate among all populations, or a square matrix of order equal to the number of populations. The last matrix has migration rate values for each pair of populations (migration['from', 'to']). Alternatively, this parameter could be a list in which the first element is a vector displaying the 't' generations that have a change of migration pattern. The following 't+1' elements are the migration rates before the change of migration pattern (single values and/or square matrices). A value of $m_{ij} = 0$ means no migration and thus no gene flow between populations $i$ and $j$ , while a value of 0.5 means random migration (and hence random reproduction) between them. This parameter is ignored for single population simulations. The default value is NULL with no migration among populations.
<code>mutation.rate</code>	A numerical value setting the mutation rate per site. It is currently restricted to biallelic SNPs (genetic structures with values 1 or 2). The default value is NULL.
<code>param.z</code>	A list with the parameter values for the function computing phenotypes. The list of parameters for each population should be included as a list of list (see Example)
<code>param.w</code>	A list with the parameter values for the fitness function. The list of parameters for each population should be included as a list of list (see Example)
<code>fun</code>	A character vector with the names of the custom phenotype and fitness functions. The default value is NULL, but it is required for the 'custom' type of evolution
<code>admixture</code>	A list with four values. The first argument is a vector of integers defining the taxon identifier ('taxa') of each population (see Example). The default value is NULL. The second argument is a logical value. If TRUE the admixture between taxa is density-dependent (see Details). The default value is FALSE. The third argument is a logical value representing equivalent migration between taxa. If TRUE the resulting individuals migrating between taxa are equivalent to the number of migrants sent by the smaller population. The default value is FALSE. The last argument is an integer defining the minimum population size for starting the admixture between different taxa. The default value is 10.

## Details

This function returns a list of populations composed of two-dimensional arrays that represents a pair of homologous chromosomes. Rows represent individuals and columns the different loci. Each element of the array is an integer defining the copy of a given allele at a given locus.

Different types of evolution are available for simulations:

- 'constant' a constant population size over time. There is no selection, equal sex ratio and each breeding pair generates two offspring. This case represent neutral evolution in which variation are due to recombination, mutation and migration. Variations in the population sizes are also possible due to the effect of migration.
- 'dynamic' a dynamic population size over time. This type of evolution introduces to type 'constant' an unequal sex ratio, a variable number of offspring and a density-dependent demographic effect to avoid exponential growth. A list with the parameters for the phenotype function (`param.z=list(sex.ratio)`) and for the fitness function (`param.w=list(mean.fitness, d.d)`) are required:
  - `sex.ratio`: a numerical value defining the sex ratio of populations. This value should be included within a list in 'param.z'.
  - `mean.fitness`: an integer with the mean number of offsprings generated by breeding pair. This value should be included within a list in 'param.w'.
  - `d.d`: numerical value introducing the density-dependent demographic effect to avoid exponential growth. This value should be included within a list in 'param.w'.  
The sex of individuals (1: females and 2: males) are generated by a binomial distribution with probability of being a male equal to the 'sex.ratio'. The fitness of individuals (in number of offspring) are obtained from:  $Poisson(\lambda) - N * d.d$ , in which  $\lambda$  is equal to 'mean.fitness' and  $N$  represents the population size.
- 'additive' additive phenotype evolution. This function introduce quantitative phenotypes and convergent or divergent selection as implemented in Quilodrán et al. (2019). A list with the parameters for the phenotype function (`param.z=list(sex.ratio, fitness.pos, bvs, add.loci, e.v)`) and for the fitness function (`param.w=list(b0,b1,b2,b3, d.v, add.loci)`) are required.

The following parameters are needed for the computation of phenotypes ( $z$ ):

- `sex.ratio`: A numerical value defining the sex ratio of populations. This value should be included within a list in 'param.z'.
- `fitness.pos`: A vector with the position of the additive loci participating in the computation of phenotypes. This value should be included within a list in 'param.z'.
- `bvs`: A matrix with the breeding values of alleles on each loci. The number of rows is equal to the number of additive loci, while the number of columns is equal to the maximum number of alleles in a locus. This object should be included within a list in 'param.z'.
- `add.loci`: An integer with the total number of additive loci participating in the computation of phenotypes. This object should be included within a list in 'param.z'.
- `e.v`: A numerical value defining a stochastic environmental variant in the computation of phenotypes. This value should be included within a list in 'param.z'.  
The default phenotype function ( $z$ ) focus on an additive genetic genotype-phenotype map. Therefore, the sum of values of alleles at each locus gives a breeding value ( $b_v$ ) for each individual at a given locus. The sum of breeding values across loci gives a breeding value for the phenotypes ( $z$ ), which is computed as follows:

$$z = \sum_{v=1}^{n_a} b_v + \varepsilon_e(0, \sigma_e)$$

Where  $n_a$  is equal to 'add.loci' and  $\sigma_e$  is equal to 'e.v'. The environmental contribution  $\varepsilon_e$  is assumed to be stochastic and normally distributed, with a mean of 0 and standard variation 'e.v'. This function returns a data.frame with rows equal to the number of individuals and two columns ('sex' and 'z')

The default fitness function ( $\omega$ ) computes a Gaussian relationship between  $z$  and  $\omega$ . The following parameters are needed:

- b0: A numerical value defining the maximum number of offspring generated by breeding pair. This value should be included within a list in 'param.w'.
- b1: A numerical value defining the phenotypic optima. In the gaussian relationship between  $z$  and  $\omega$ , the value of 'b1' represent the  $z$  value expected to produce the maximum number of offspring. This value should be included within a list in 'param.w'. The difference in phenotypic optima between the populations drives the strength of 'divergent selection'. Populations exposed to equal phenotypic optima are considered to be under 'concordant selection'.
- b2: A numerical value defining the variance of the Gaussian curve. This value should be included within a list in 'param.z'.
- b3: A numerical value defining the intensity of the density-dependence on the fitness of individuals in a population of size  $N$ . This value should be included within a list in 'param.w'.
- d.v: A numerical value defining a stochastic demographic variant in the fitness of individuals. This value should be included within a list in 'param.w'.
- add.loci: An integer with the total number of additive loci participating in the computation of phenotypes. This object should be included within a list in 'param.w'.

The default fitness function ( $\omega$ ) has the form:

$$\omega = b_0 \exp^{-\frac{1}{2} \left( \frac{z - b_1 n_a}{b_2 n_a} \right)^2} - b_3 N + \varepsilon_d(0, \sigma_d)$$

Where  $n_a$  is equal to 'add.loci',  $N$  is the population size and  $\sigma_d$  is equal to 'd.v'. The demographic variant  $\varepsilon_d$  is assumed to be stochastic and normally distributed, with a mean of 0 and standard variation 'd.v'. This function returns a vector with the fitness value ( $\omega$ ) of individuals.

- 'custom' is a custom computation of phenotypes and fitness functions. These functions will define the type of selection fitting particular case studies. The name of each user defined function should be introduced in the 'fun' parameter as a character vector with two elements e.g. c('phenotype', 'fitness'). Assuming a per-generation time step, the potential number of offspring produced by each individual depends on its phenotype  $\omega = f(z)$ , which in turn depends on the individual genotype and on the environment  $z = g(G, E)$ .  $G$  is a numeric value determined by an individual's genotype, representing the genetic value of the genotype. In the case of an additive genetic map, the genetic value of a genotype will be a breeding value.  $E$  represents the effect of the environment on phenotypic expression, and this enables the effects of plasticity on phenotypic expression to be captured. The list of parameters 'param.z' and 'param.w' include all needed parameters for the custom 'phenotype' and 'fitness' functions. These lists should not include variables. For the phenotype function, the variable genetic structure of individuals is already included as the object 'struct', which should also be the first argument of the custom phenotype function, followed by "...". This means that all parameters used in the custom phenotype function are only included in 'param.z'. The custom phenotype function should return a matrix with two columns, named "z" and "sex". The first column is the resulting phenotype value and the second column represents the assignation of sex to each individual. For the fitness function, the variable individual phenotype value, sex of individuals and the population size are already included in the environment. The function should start with these three elements ("z","sex","n"), followed by "...". The user does not need to

use all three of these variables. All parameters needed for the custom fitness function should be included in 'param.w'. This function should return a vector 'w' with fitness values for the individuals (see Examples).

The recombination between homologous chromosomes are either of type 'map' or 'average'. The first case needs a vector with the recombination rate ( $\rho$ ) between neighbour loci of length equal to  $nl - 1$  ( $nl$ : number of loci). The probability of having a crossover (1) or not (0) is uniformly distributed at a rate defined by the value of  $\rho$  between loci (i.e. positions with a probability smaller than  $\rho$  recombine). The uniform distribution allows each position with the same values of  $\rho$  to have an equal chance of crossover across all iterations. There is no recombination between homologous chromosomes when  $\rho = 0$ , both loci are completely linked (e.g. within an inversion or situated close to centromeres), while with a value of  $\rho = 0.5$ , the recombination rate is completely random (i.e. both loci are very distant on the same chromosome or are located on different chromosomes). A value of  $\rho < 0.5$  means the loci are physically linked. The second case, when recombination is of type 'average', a numerical value with the average recombination rate per base pair should be supplied, the loci position and the size of the chromosome in megabase are also required. The crossover points are exponentially distributed as a Poisson process (see Example).

### Value

A list of objects of class "struct" or array. The initial and final genetic diversity are recorded on the list. The outputs of additional generations are recorded when specified by the parameter 'time'.

### References

Quilodrán, C. S., Ruegg, K., Sendell-Price, A. T., Anderson, E., Coulson, T. and Clegg, S. (2020). The many population genetic and demographic routes to islands of genomic divergence. *Methods in Ecology and Evolution* 11(1):6-21.. doi:10.1111/2041210X.13324.

### See Also

[initial.struct](#)

### Examples

```
## Not run:
## We start with a population of 20 individuals and 10 biallelic SNPs
initial.population.size <- 20
n.loci <- 10
n.alleles.per.locus <- 2

start1 <- initial.struct(initial.population.size,n.loci,n.alleles.per.locus)

#####
# Type of evolution = "constant" #
#####

## We set a recombination map and the number of generations to be simulated
recom.map <- rep(0.5, n.loci-1) #all loci are independent
n.gens <- 10
pop <- evolve(list(start1), n.gens, "constant", "map", recom.map)

## A similar simulation but with recombination type "average".
## We need to specify the position of loci and the size of the chromosome (MB)
loci.pos<- sample(80000: 1200000, 10) #random loci positions
```

```

chromo_mb<-12000000 #chromosome size
crossover <- 1/100000000.0 #average recombination rate (1 cM/MB)
pop <- evolve(list(start1), n.gens, "constant", "average", crossover, loci.pos, chromo_mb)

# We include a mutation rate for the simulation of biallelic loci
pop <- evolve(list(start1), n.gens, "constant", "average", crossover,
              loci.pos, chromo_mb, mutation.rate = 0.0001)

# A second population is included to incorporate the effect of migration
start2 <- initial.struct(initial.population.size,n.loci,n.alleles.per.locus)
pop <- evolve(list(start1, start2), n.gens, "constant", "average", crossover,
              loci.pos, chromo_mb, migration.rate = 0.01)

# The sex of individuals may be set for the starting generations.
sex.start1 <- sample(1:2, initial.population.size, replace=T)
sex.start2 <- sample(1:2, initial.population.size, replace=T)
init.sex <- list(sex.start1, sex.start2)
pop <- evolve(list(start1, start2), n.gens, "constant", "average", crossover,
              loci.pos, chromo_mb, init.sex = init.sex)

#####
# Type of evolution = "dynamic" #
#####

## We set the parameters for the computation of phenotypes and for the
## fitness function of the type 'dynamic'
sex.ratio <- 0.5
mean.fitness <- 3 #mean number of offsprings per breeding pair
d.d <- 0.01 #density-dependent demographic effect

set.seed(1) #setting the seed for reproducible random numbers
pop <- evolve(list(start1), n.gens, "dynamic", "map", recom.map, param.z = list(list(sex.ratio)),
              param.w = list(list(mean.fitness, d.d)))

#####
# Type of evolution = "additive" #
#####

## We set the parameters for the computation of phenotypes and for
## the fitness function of the type 'additive'
sex.ratio <- 0.5
fitness.pos <- 1:n.loci #all simulated loci are additive
add.loci <- n.loci

# next line is breeding value of additive loci
bvs <- t(array( seq(0,1, length = n.alleles.per.locus) ,c(n.alleles.per.locus, add.loci)))

e.v <- 0.01 # stochastic environmental variant
param.z <- list(sex.ratio, fitness.pos, bvs, add.loci, e.v)

b0 <- 6 # maximum number of offspring
b1 <- 10 # phenotypic optima
b2 <- 4 # variance of the Gaussian curve
b3 <- 0.01 # intensity of the density-dependence
d.v = 1 # stochastic demographic variant
param.w <- list(b0,b1,b2,b3, d.v)

```

```

set.seed(1) #setting the seed for reproducible random numbers
pop<-evolve(list(start1), n.gens, "additive", "map", recom.map,
            param.z = list(param.z), param.w = list(param.w))

#####
# Type of evolution = "custom" #
#####

## We set a custom 'phenotype' and 'fitness' function.
## In this example, the custom functions are very similar to the default 'additive' ones.

phenotype2 <- function(struct, ...){
  pop.struct <- struct[ , fitness.pos, ]
  temp <- dim(pop.struct)
  mat <- matrix(1:temp[2],temp[1],temp[2],byrow=TRUE)

  # the next line gives an array that gives the locus index at each position in pop.struct
  loci.n <- array(mat,c(temp[1],temp[2],2))

  new.n <- (pop.struct-1)*add.loci+loci.n
  bvv <- as.vector(bvs) # turn to bvs
  outp <- array(bvv[new.n],c(temp[1],temp[2],2))
  z <- apply(outp,1,sum)+rnorm(temp[1],0,e.v)
  sex <- rbinom(nrow(struct), 1, sex.ratio)+1
  return(cbind(z = z, sex = sex))
}

fitness2 <- function(z, sex, n, ...){
  a <- b0
  b <- b1
  c <- b2
  w <- round( a*exp(-(z-b)^2)/(2*(c)^2) ) - b3*n + rnorm(n,0,d.v) , 0)
  w <- ifelse(w<0,0,w)
  return(w)
}

set.seed(1) #setting the seed for reproducible random numbers
pop<-evolve(list(start1), n.gens, "custom", "map", recom.map,
            param.z =list(sex.ratio = sex.ratio, fitness.pos = fitness.pos,
                          bvs = bvs, add.loci = add.loci, e.v = e.v),
            param.w = list(b0 = b0, b1 = b1, b2 = b2, b3 = b3, d.v = d.v),
            fun=c("phenotype2", "fitness2"))

#####
# Various populations with change of migration patterns over time #
#           output recorded each 10 generations           #
#####

initial.population.size <- 100
n.loci <- 10
n.alleles.per.locus <- 2
recom.map <- rep(0.5, n.loci-1) #all loci are independent
n.gens <- 100 #total number of simulated generations
n.gens.out <- 10 #output recorded each 10 generations
sex.ratio <- 0.5
mean.fitness <- 4 #mean number of offsprings per breeding pair

```

```

d.d <- 0.001 #density-dependent demographic effect
set.seed(1)
start1 <- initial.struct(initial.population.size,n.loci,n.alleles.per.locus)
start2 <- array(NA, dim = c(0, n.loci, 2)) #an empty population
start3 <- initial.struct(2,n.loci,n.alleles.per.locus) #a population with two individuals
struct <- list(start1, start2, start3) #initial population structure

#Parameters of type 'dynamics'
param.z1 <- list(sex.ratio)
param.w1 <- list(mean.fitness, d.d)

npop <- length(struct) #number of populations
mr1 <- 0.01 #initial migration rate
mr2 <- matrix(rep(0.02, npop*npop), nrow = npop, ncol = npop, byrow = TRUE); diag(mr2) <- 1
mr3 <- 0
mchange <- c(10, 20) #a change of migration pattern occurs after these generations

pop <- evolve(list(start1, start2, start3), c(n.gens, n.gens.out), "dynamic",
             "map", recom.map, param.z = list(param.z1, param.z1, param.z1),
             param.w = list(param.w1,param.w1,param.w1),
             migration.rate = list(mchange, mr1, mr2, mr3))

#####
# Admixture between three taxa inhabiting two demes #
# six populations in total #
#####

n.loci <- 10
recom.map <- rep(0.5, n.loci-1) #all loci are independent
n.gens <- 100 #total number of simulated generations
sex.ratio <- 0.5
mean.fitness <- 4 #mean number of offsprings per breeding pair
d.d <- 0.01 #density-dependent demographic effect

set.seed(1)
#The taxa are differentiated at positions 3 and 4
#N=100 | 10 loci | variable number of alleles (average 2) | representing taxon 1
start1 <- initial.struct(N=100,nl=n.loci,na=rpois(n.loci, 1) +1,taxon = list(pos=3:4, id=1))
#an empty population
start2 <- array(NA, dim = c(0, n.loci, 2))
#N=100 | 10 loci | two alleles in all loci | representing taxon 2
start3 <- initial.struct(N=10,nl=n.loci,na=2,taxon = list(pos=3:4, id=2))
#N=80 | 10 loci | two alleles in all loci | representing taxon 2
start4 <- initial.struct(N=80,nl=n.loci,na=2,taxon = list(pos=3:4, id=2))
#N=100 | 10 loci | variable number of alleles (average 5) | representing taxon 3
start5 <- initial.struct(N=100,nl=n.loci,na=rpois(n.loci, 5) +1,taxon = list(pos=3:4, id=3))
#an empty population
start6 <- array(NA, dim = c(0, n.loci, 2))

#initial population structure
struct <- list(start1, start2, start3, start4, start5, start6)

#Taxa id for each population
taxa.id <- c(1,1,2,2,3,3)

#A different migration pattern between taxa and within taxon

```

```

wt <- 0.01 # within taxon
bt <- 0.001 # between taxa
npop <- length(struct)

migration.rate <- matrix(
  c(c(1,wt,bt,bt,bt,bt),
    c(wt,1,bt,bt,bt,bt),
    c(bt,bt,1,wt,bt,bt),
    c(bt,bt,wt,1,bt,bt),
    c(bt,bt,bt,bt,1,wt),
    c(bt,bt,bt,bt,wt,1) ), nrow=npop)
#migration.rate <- mmatrix(migration.rate, npop) #alternatively you may use the function mmatrix()

#Parameters of type 'dynamics'
param.z1 <- list(sex.ratio)
param.w1 <- list(mean.fitness, d.d)

set.seed(1)
pop <- evolve(struct, n.gens, "dynamic", "map", recom.map, param.z = list(param.z1,
  param.z1, param.z1, param.z1, param.z1),
  param.w = list(param.w1,param.w1,param.w1, param.w1,param.w1,param.w1),
  migration.rate = migration.rate, admixture = list(taxa=taxa.id, dd=TRUE, nmin=20))

#####
# Admixture between two taxa inhabiting three demes #
#           six populations in total                #
#           One locus fixed to each population      #
#####

n.loci <- 10 + 1 #one locus will be used to recognize populations
recom.map <- rep(0.5, n.loci-1) #all loci are independent
n.gens <- 100 #total number of simulated generations
sex.ratio <- 0.5
mean.fitness <- 4 #mean number of offsprings per breeding pair
d.d <- 0.01 #density-dependent demographic effect

set.seed(1)
#Two taxa in three demes | six populations in total
#N=10 | 10 loci + 1 loci fixed to each population | biallelic | taxon 1 | population 1 | deme 1
start1 <- initial.struct(N=10,nl=n.loci,na=2,
  taxon = list(pos=c(1,3:5), id=c(pop=1,alleles=rep(1,3))))
#N=10 | 10 loci + 1 loci fixed to each population | biallelic | taxon 1 | population 2 | deme 2
start2 <- initial.struct(N=10,nl=n.loci,na=2,
  taxon = list(pos=c(1,3:5), id=c(pop=2,alleles=rep(1,3))))
#N=10 | 10 loci + 1 loci fixed to each population | biallelic | taxon 1 | population 3 | deme 3
start3 <- initial.struct(N=10,nl=n.loci,na=2,
  taxon = list(pos=c(1,3:5), id=c(pop=3,alleles=rep(1,3))))
#N=100 | 10 loci + 1 loci fixed to each population | biallelic | taxon 1 | population 4 | deme 1
start4 <- initial.struct(N=100,nl=n.loci,na=2,
  taxon = list(pos=c(1,3:5), id=c(pop=4,alleles=rep(2,3))))
#N=100 | 10 loci + 1 loci fixed to each population | biallelic | taxon 1 | population 5 | deme 2
start5 <- initial.struct(N=100,nl=n.loci,na=2,
  taxon = list(pos=c(1,3:5), id=c(pop=5,alleles=rep(2,3))))
#N=100 | 10 loci + 1 loci fixed to each population | biallelic | taxon 1 | population 6 | deme 3
start6 <- initial.struct(N=100,nl=n.loci,na=2,
  taxon = list(pos=c(1,3:5), id=c(pop=6,alleles=rep(2,3))))

```

```

#initial population structure
struct <- list(start1, start2, start3, start4, start5, start6)

#Taxa id for each population
taxa.id <- c(1,1,1,2,2,2)

#A different migration pattern between taxa and within taxon
wt <- 0.01 # within taxon
bt <- 0.001 # between taxa
npop <- length(struct)

migration.rate <- mmatrix(iffelse(outer(taxa.id, taxa.id, "==")==T, wt, bt), npop)[,,1]

#Parameters of type 'dynamics'
param.z1 <- list(sex.ratio)
param.w1 <- list(mean.fitness, d.d)

set.seed(1)
pop <- evolve(struct, n.gens, "dynamic", "map", recom.map,
              param.z = list(param.z1,param.z1, param.z1, param.z1, param.z1, param.z1),
              param.w = list(param.w1,param.w1,param.w1, param.w1,param.w1,param.w1),
              migration.rate = migration.rate, admixture = list(taxa=taxa.id, dd=TRUE, nmin=20))

#####
#       Two different fitness function per taxa       #
#           Types constant and dynamic                #
# One taxon reach extinction before the simulation end #
#####

# structct: objects of class "struct" with the genetic structure of populations.
#           It is allocated to the simulation environment.
phenotypeCD <- function(struct, type = c("constant", "dynamic"), ...){

  switch(type,
    constant = {
      n <- nrow(struct)
      if(n %% 2 == 0){
        sex <- as.data.frame(cbind(sex =sample(rep(0:1,each=n/2))+1)) #zs
        rownames(sex)=rownames(struct)
      } else {
        neven = floor(n) + floor(n) %% 2 -2
        sex <- as.data.frame(cbind(sex =c(sample(rep(0:1,each=neven/2)), sample(0:1,1))+1)) #zs
        rownames(sex)=rownames(struct)
      }
    },
    dynamic = {
      sex.ratio <- sex.ratio
      n <- nrow(struct)
      sex <- as.data.frame(cbind(sex = rbinom(n, 1, sex.ratio)+1)) #zs
      rownames(sex)=rownames(struct)
    }
  )
  if(nrow(sex)==0){ sex[1,1] <-NA }

  return(cbind(z = NA, sex = sex))
}

```

```

# time.t: running time of the simulation. It is allocated to the simulation environment
# n: population size. It is allocated to the simulation environment
# t.w.change and newfit are two new variables defining the time the fitness change
#   and the new fitness value, respectively.
fitnessCD <- function(time.t, n, type = c("constant", "dynamic"), t.w.change=50, newfit=0, ...){

switch(type,
  constant = {
    mean.fitness <- 2
    if(time.t>t.w.change) {mean.fitness<-newfit}
    w <- rep(mean.fitness, n)
  },
  dynamic = {
    mean.fitness <- mean.fitness
    d.d <- d.d
    w <- round( pmax(rpois(n, lambda=mean.fitness) - d.d*n, 0) )
  })
return(w)
}

param.z1 <- param.z2 <-param.z3 <- list(sex.ratio = sex.ratio, type = "dynamic")
param.w1 <- param.w2 <- param.w3 <- list(mean.fitness = mean.fitness, d.d = d.d, type = "dynamic")

param.z4 <- param.z5 <-param.z6 <- list(type = "constant")
param.w4 <- param.w5 <-param.w6 <- list(type = "constant")

set.seed(1)
#output each 5 generations
pop <- evolve(struct, c(n.gens, 5), "custom", "map", recom.map,
  param.z = list(param.z1,param.z2, param.z3, param.z4, param.z5, param.z6),
  param.w = list(param.w1,param.w2,param.w3, param.w4,param.w5,param.w6),
  migration.rate = migration.rate, admixture = list(taxa=taxa.id, dd=TRUE, nmin=20),
  fun=c("phenotypeCD", "fitnessCD"))

#####
# Plotting population sizes #
#####

npop=6
n.outputs=21 #each 5 generations during 100 generations
popsizes<-sapply(1:npop, function(i){
sapply(1:n.outputs, function(j){
  dim(pop[[j]][[i]])[1]
})
})

plot(1:21, popsizes[,1], type="l", bty="l", ylab="Population size",
  xlab="generations", xaxs="i", yaxs="i", las=1, ylim=c(0, 310), lwd=2, col="darkred")
lines(1:21, popsizes[,2], col="darkgoldenrod", lwd=2)
lines(1:21, popsizes[,3], col="darkorange", lwd=2)
lines(1:21, popsizes[,4], col="darkblue", lwd=2)
lines(1:21, popsizes[,5], col="darkolivegreen4", lwd=2)
lines(1:21, popsizes[,6], col="darkturquoise", lwd=2)

legend("bottomright", legend=paste("pop", 1:6, sep=""),
  col=c("red", "goldenrod", "darkorange", "blue", "green", "turquoise"), lwd=2, bty="n")

```

```
## End(Not run)
```

---

fitness	<i>Fitness function</i>
---------	-------------------------

---

### Description

This function computes a fitness value ( $\omega$ ) depending on the phenotype ( $z$ ) of individuals. The relationship between both variables is assumed to be Gaussian.

### Usage

```
fitness(z, N, b0, b1, b2, b3, d.v)
```

### Arguments

<code>z</code>	A data.frame with the phenotype value ('z') and the sex of each individual in a population.
<code>N</code>	Population size of the population.
<code>b0</code>	A numerical value defining the maximum number of offspring that may be generated by a breeding pair.
<code>b1</code>	A numerical value defining the phenotypic optima of a given population (see Details).
<code>b2</code>	A numerical value defining the variance of the Gaussian curve.
<code>b3</code>	A numerical value defining the intensity of the density-dependence on the fitness of individuals in a population of size 'N'.
<code>d.v</code>	A numerical value defining a stochastic demographic variant in the fitness of individuals.

### Details

This function is used internally in the function `evolve()` of type 'selection' in order to compute the fitness value of individuals.

The value of 'b1' represents the  $z$  value expected to produce the maximum number of offspring. The difference in phenotypic optima between the populations drives the strength of 'divergent selection'. Populations exposed to equal phenotypic optima are considered to be under 'concordant selection'.

This fitness function ( $\omega$ ) has the following form:

$$\omega = b_0 \exp^{-\frac{1}{2} \left( \frac{z-b_1}{b_2} \right)^2} - b_3 N + \varepsilon_d(0, \sigma_d)$$

Where  $n_a$  is equal to 'add.loci',  $N$  is the population size and  $\sigma_d$  is equal to 'd.v'. The demographic variant  $\varepsilon_d$  is assumed to be stochastic and normally distributed, with a mean of 0 and standard variation 'd.v'.

### Value

A vector with the fitness value ( $\omega$ ) for each individual.

## References

Quilodr an, C. S., Ruegg, K., Sendell-Price, A. T., Anderson, E., Coulson, T. and Clegg, S. (2020). The multiple population genetic and demographic routes to islands of genomic divergence. *Methods in Ecology and Evolution*. doi:10.1111/2041210X.13324.

## See Also

[evolve phenotype](#)

## Examples

```
## We first create a random population with 100 individuals and 10 loci
N <- 100 # Population size
nl <- 10 # Number of additive loci
na <- 4 # Number of alleles per locus
G <- initial.struct(N,nl,na)

## Additional parameters are needed for the computation of phenotypes
bvs <- t(array( seq(0,1, length = na) ,c(na, nl)))
sex.ratio <- 0.5
e.v=0.01

## Now we compute the additive phenotype value of individuals
phen <- phenotype(G, bvs, nl, sex.ratio, e.v)

## Additional parameters are needed for the fitness function
b0 <- 6
b1 <- 0.25
b2 <- 0.5
b3 <- 0.01
d.v = 1

## The fitness values of individuals are computed as follows:
fitness(phen, N, b0, b1, b2, b3, d.v)
```

---

initial.struct

*Initial Structure*

---

## Description

This function generates an object of class "struct" with an initial genetic population structure for simulations.

## Usage

```
initial.struct(N, nl, na, taxon = list(pos = NULL, id = NULL))
```

## Arguments

N	Number of individuals in the initial population
nl	Number of simulated loci

na	Number of alleles at each locus. This parameter is either a single value setting the number of alleles for all loci or a vector with the number of alleles at each loci.
taxon	a list of two vectors providing the fixed positions 'pos' that differentiate the taxon and the allelic 'id' for these positions. The 'id' could also be a single value repeated on all fixed positions. The default value is NULL.

### Details

This function returns a three-dimensional array. Rows represent individuals and columns the different loci. Each element of the array is an integer defining the copy of a given allele at a given locus. The third dimension of the array has two layers representing a pair of homologous chromosomes.

### Value

An object of class "struct" or an array.

### See Also

[evolve](#)

### Examples

```
##Initial population size of 10 individuals with 5 biallelic loci
initial.population.size=10
n.loci=5
n.alleles.per.locus=2
initial.struct(initial.population.size,n.loci,n.alleles.per.locus)

##A differentiated taxon with fixed alleles in two loci
initial.struct(initial.population.size,n.loci,n.alleles.per.locus, taxon = list(pos=3:4, id=1:2))
```

---

phenotype	<i>Phenotype function</i>
-----------	---------------------------

---

### Description

A function for the computation of additive phenotypes

### Usage

```
phenotype(G, bvs, add.loci, sex.ratio, e.v)
```

### Arguments

G	An object of class "struct" with the genetic structure of each individual in the population.
bvs	A matrix with the breeding values of alleles on each loci. The number of rows is equal to the number of additive loci, while the number of columns is equal to the maximum number of alleles in a locus.
add.loci	An integer with the total number of additive loci participating in the computation of phenotypes.

sex.ratio	A numerical value defining the sex ratio of populations
e.v	A numerical value defining a stochastic environmental variant in the computation of phenotypes.

### Details

This function is used internally in the function `evolve()` of type 'selection' in order to compute the additive phenotype value of individuals.

This phenotype function ( $z$ ) focuses on an additive genetic genotype-phenotype map. The sum of values of alleles at each locus gives a breeding value ( $b_v$ ) for each individual at a given locus. The sum of breeding values  $bvs$  across loci gives a breeding value for the phenotypes ( $z$ ), which is computed as follows:

$$z = \sum_{v=1}^{n_a} b_v + \varepsilon_e(0, \sigma_e)$$

Where  $n_a$  is equal to 'add.loci' and  $\sigma_e$  is equal to 'e.v'. The environmental contribution  $\varepsilon_e$  is assumed to be stochastic and normally distributed, with a mean of 0 and standard variation 'e.v'.

### Value

A data.frame with rows equal to the number of individuals and two columns ('sex' and 'z')

### References

Quilodrán, C. S., Ruegg, K., Sendell-Price, A. T., Anderson, E., Coulson, T. and Clegg, S. (2020). The multiple population genetic and demographic routes to islands of genomic divergence. *Methods in Ecology and Evolution*. doi:10.1111/2041210X.13324.

### See Also

[evolve fitness](#)

### Examples

```
## We first create a random population with 100 individuals and 10 loci
N <- 100 # Population size
nl <- 10 # Number of additive loci
na <- 4 # Number of alleles per locus
G <- initial.struct(N,nl,na)

## Additional parameters are needed for the computation of phenotypes
bvs <- t(array( seq(0,1, length = na) ,c(na, nl)))
sex.ratio <- 0.5
e.v=0.01

## Now we compute the additive phenotype value of individuals
phenotype(G, bvs, nl, sex.ratio, e.v)
```

---

`struct2pegas`*Conversion of class struct to class loci*

---

**Description**

This function converts an object of class "struct" to an object of class "loci" that can be used by the package pegas

**Usage**

```
struct2pegas(x)
```

**Arguments**

`x` List of objects of class "struct" with the initial genetic structure of populations.

**Examples**

```
## We first create a random population with 100 individuals and 10 biallelic loci
N <- 100 # Population size
nl <- 10 # Number of additive loci
na <- 2 # Number of alleles per locus
G <- initial.struct(N,nl,na)
```

```
## We convert the object of class 'struct' into 'loci'
struct2pegas(list(G))
```

# Index

evolve, [2](#), [14–16](#)

fitness, [13](#), [16](#)

glads (glads-package), [2](#)

glads-package, [2](#)

initial.struct, [6](#), [14](#)

phenotype, [14](#), [15](#)

struct2pegas, [17](#)