simuPOP Reference Manual

Release 0.8.8 (Rev: 1943)

Bo Peng

December 2004

Last modified
November 18, 2008

Department of Epidemiology, U.T. M.D. Anderson Cancer Center
Email: bpeng@mdanderson.org
URL: http://simupop.sourceforge.net
Mailing List: simupop-list@lists.sourceforge.net

Abstract

simuPOP is a forward-time population genetics simulation environment. Unlike coalescent-based programs, simuPOP evolves populations forward in time, subject to arbitrary number of genetic and environmental forces such as mutation, recombination, migration and population/subpopulation size changes. Statistics of populations can be calculated and visualized dynamically which makes simuPOP an ideal tool to demonstrate population genetics models; generate datasets under various evolutionary settings, and more importantly, study complex evolutionary processes and evaluate gene mapping methods.

simuPOP is written in C++ and is provided as Python modules. It provides a large number of building blocks (populations, mating schemes, various genetic forces in the form of operators, simulators and gene mapping methods) to construct a simulation. This provides a R/Splus or Matlab-like environment where users can interactively create, manipulate and evolve populations, monitor and visualize population statistics and apply gene mapping methods. Please refer to the *simuPOP user's guide* for a detailed introduction to simuPOP concepts, and a number of examples on how to use simuPOP to perform various simulations.

This reference manual lists all variables, functions, classes and utility modules of simuPOP. Please report any error to the simuPOP mailing list `simupop-list@lists.sourceforge.net`.

How to cite simuPOP:

> Bo Peng and Marek Kimmel (2005) simuPOP: a forward-time population genetics simulation environment. *bioinformatics*, **21** (18): 3686-3687.
>
> Bo Peng and Christopher Amos (2008) Forward-time simulations of nonrandom mating populations using simuPOP. *bioinformatics*, **24** (11)" 1408-1409.

CONTENTS

CHAPTER

ONE

simuPOP Components

1.1 Individual and Population

1.1.1 Class `GenoStruTrait`

All individuals in a population share the same genotypic properties such as number of chromosomes, number and position of loci, names of markers, chromosomes, and information fields. These properties are stored in this `GenoStruTrait` class and are accessible from `individual`, `population`, and `simulator` classes. Currently, a genotypic structure consists of

- Ploidy, namely the number of homologous sets of chromosomes, of a population. Haplodiploid population is also supported.
- Number of chromosomes and number of loci on each chromosome.
- Positions of loci, which determine the relative distance between loci on the same chromosome. No unit is assumed so these positions can be ordinal (`1`, `2`, `3`, ..., the default), in physical distance (`bp`, `kb` or `mb`), or in map distance (e.g. centiMorgan) depending on applications.
- Names of alleles. Although alleles at different loci usually have different names, simuPOP uses the same names for alleles across loci for simplicity.
- Names of loci and chromosomes.
- Names of information fields attached to each individual.

In addition to basic property access functions, this class also provides some utility functions such as `locusByName`, which looks up a locus by its name.

class `GenoStruTrait`()

A `GenoStruTrait` object is created with the creation of a `population` so it cannot be initialized directly.

`ploidy`()

Return the number of homologous sets of chromosomes, specified by the *ploidy* parameter of the `population` function. Return 2 for a haplodiploid population because two sets of chromosomes are stored for both males and females in such a population.

`ploidyName`()

Return the ploidy name of this population, can be one of `haploid`, `diploid`, `haplodiploid`, `triploid`, `tetraploid` or `#-ploid` where `#` is the ploidy number.

`chromBegin`(*chrom*)

Return the index of the first locus on chromosome *chrom*.

chromByName(*name*)
Return the index of a chromosome by its *name*.

chromEnd(*chrom*)
Return the index of the last locus on chromosome *chrom* plus 1.

chromName(*chrom*)
Return the name of a chromosome *chrom*. Default to `chrom#` where # is the 1-based index of the chromosome.

chromNames()
Return a list of the names of all chromosomes.

chromType(*chrom*)
Return the type of a chromosome *chrom* (`1` for `Autosome`, `2` for `ChromosomeX`, `3` for `ChromosomeY`, and `4` for `Mitochondrial`).

chromTypes()
Return the type of all chromosomes (`1` for `Autosome`, `2` for `ChromosomeX`, `3` for `ChromosomeY`, and `4` for `Mitochondrial`).

numChrom()
Return the number of chromosomes.

absLocusIndex(*chrom, locus*)
Return the absolute index of locus *locus* on chromosome *chrom*. An `IndexError` will be raised if *chrom* or *locus* is out of range. c.f. `chromLocusPair`.

chromLocusPair(*locus*)
Return the chromosome and relative index of a locus using its absolute index *locus*. c.f. `absLocusIndex`.

lociByNames(*names*)
Return the indexes of loci with names *names*. Raise a `ValueError` if any of the loci cannot be found.

lociDist(*loc1, loc2*)
Return the distance between loci *loc1* and *loc2* on the same chromosome. A negative value will be returned if *loc1* is after *loc2*.

lociNames()
Return the names of all loci specified by the *lociNames* parameter of the `population` function.

lociPos()
Return the positions of all loci, specified by the *lociPos* prameter of the `population` function. The default positions are 1, 2, 3, 4, ... on each chromosome.

locusByName(*name*)
Return the index of a locus with name *name*. Raise a `ValueError` if no locus is found.

locusName(*loc*)
Return the name of locus *loc* specified by the *lociNames* parameter of the `population` function. Default to `locX-Y` where `X` and `Y` are 1-based chromosome and locus indexes (`loc1-1`, `loc1-2`, ... etc)

locusPos(*loc*)
Return the position of locus *loc* specified by the *lociPos* parameter of the `population` function. An `IndexError` will be raised if the absolute index *loc* is greater than or equal to the total number of loci.

numLoci(*chrom*)
Return the number of loci on chromosome *chrom*, equivalent to `numLoci()[`*chrom*`]`.

numLoci()
Return the number of loci on all chromosomes.

totNumLoci()
Return the total number of loci on all chromosomes.

alleleName(*allele*)
Return the name of allele *allele* specified by the *alleleNames* parameter of the `population` function. If the name of an allele is not specified, its index (`'0'`, `'1'`, `'2'`, etc) is returned. An `IndexError` will be raised if *allele* is larger than the maximum allowed allele state of this module (`MaxAllele()`).

alleleNames()
Return a list of allele names given by the *alleleNames* parameter of the `population` function. This list does not have to cover all possible allele states of a population so `alleleNames()[`*allele*`]` might fail (use `alleleNames(`*allele*`)` instead).

infoField(*idx*)
Return the name of information field *idx*.

infoFields()
Return a list of the names of all information fields of the population.

infoIdx(*name*)
Return the index of information field *name*. Raise an `IndexError` if *name* is not one of the information fields.

1.1.2 Class `individual`

A `population` consists of individuals with the same genotypic structure. An `individual` object cannot be created independently, but refences to inidividuals can be retrieved using member functions of a `population` object. In addition to structural information shared by all individuals in a population (provided by class `genoStruTrait`), an `individual` class provides member functions to get and set *genotype*, *sex*, *affection status* and *information fields* of an individual.

Genotypes of an individual are stored sequentially and can be accessed locus by locus, or in batch. The alleles are arranged by position, chromosome and ploidy. That is to say, the first allele on the first chromosome of the first homologous set is followed by alleles at other loci on the same chromsome, then markers on the second and later chromosomes, followed by alleles on the second homologous set of the chromosomes for a diploid individual. A consequence of this memory layout is that alleles at the same locus of a non-haploid individual are separated by `individual::totNumLoci()` loci. It is worth noting that access to invalid chromosomes, such as the Y chromosomes of female individuals, are not restricted.

class individual()
An `individual` object cannot be created directly. It has to be accessed from a `population` object using functions such as `population::individual(`*idx*`)`.

allele(*idx*)
Return the current allele at a locus, using its absolute index *idx*.

allele(*idx, p*)
Return the current allele at locus *idx* on the *p*-th set of homologous chromosomes.

allele(*idx, p, chrom*)
Return the current allele at locus *idx* on chromosome *chrom* of the *p*-th set of homologous chromosomes.

setAllele(*allele, idx*)
Set allele *allele* to a locus, using its absolute index *idx*.

setAllele(*allele, idx, p*)
Set allele *allele* to locus *idx* on the *p*-th homologous set of chromosomes.

setAllele(*allele, idx, p, chrom*)
Set allele *allele* to locus *idx* on chromosome *chrom* of the *p*-th homologous set of chromosomes.

genotype()
Return an editable array (a `carray` of length `totNumLoci()*ploidy()`) that represents all alleles of an individual.

genotype(*p*)
Return an editable array (a `carray` of length `totNumLoci()`) that represents all alleles on the *p*-th homologous set of chromosomes.

genotype(*p, chrom*)
Return an editable array (a `carrary` of legnth `numLoci(`*chrom*`)`) that represents all alleles on chromosome *chrom* of the *p*-th homologous set of chromosomes.

setGenotype(*geno*)
Fill the genotype of an individual using a list of alleles *geno*. `geno` will be reused if its length is less than `totNumLoci()*ploidy()`.

setGenotype(*geno, p*)
Fill the genotype of the *p*-th homologous set of chromosomes using a list of alleles *geno*. `geno` will be reused if its length is less than `totNumLoci()`.

setGenotype(*geno, p, chrom*)
Fill the genotype of chromosome *chrom* on the *p*-th homologous set of chromosomes using a list of alleles *geno*. `geno` will be reused if its length is less than `mumLoci(`*chrom*`)`.

setSex(*sex*)
Set individual sex to `Male` or `Female`.

sex()
Return the sex of an individual, `1` for male and `2` for female.

sexChar()
Return the sex of an individual, `M` for male or `F` for `female`.

affected()
Return `True` if this individual is affected.

affectedChar()
Return `A` if this individual is affected, or `U` otherwise.

setAffected(*affected*)
Set affection status to *affected* (`True` or `False`).

info(*idx*)
Return the value of an information field *idx* (an index).

info(*name*)
Return the value of an information field *name*.

intInfo(*idx*)
Return the value of an information field *idx* (an index) as an integer number.

intInfo(*name*)
Return the value of an information field *name* as an integer number.

setInfo(*value, idx*)
Set the value of an information field *idx* (an index) to *value*.

setInfo(*value, name*)
Set the value of an information field *name* to *value*.

1.1.3 Class `population`

A simuPOP population consists of individuals of the same genotypic structure, organized by generations, subpopulations and virtual subpopulations. It also contains a Python dictionary that is used to store arbitrary population variables. In addition to genotypic structured related functions provided by the `genoStruTrait` class, the population class provides a large number of member functions that can be used to

- Create, copy and compare populations.
- Manipulate subpopulations. A population can be divided into several subpopulations. Because individuals only mate with individuals within the same subpopulation, exchange of genetic information across subpopulations can only be done through migration. A number of functions are provided to access subpopulation structure information, and to merge and split subpopulations.
- Define and access virtual subpopulations. A *virtual subpopulation splitter* can be assigned to a population, which defines groups of individuals called *virtual subpopulations* (VSP) within each subpopulation.
- Access individuals individually, or through iterators that iterate through individuals in (virtual) subpopulations.
- Access genotype and information fields of individuals at the population level. From a population point of view, all genotypes are arranged sequentially individual by individual. Please refer to class `individual` for an introduction to genotype arragement of each individual.
- Store and access *ancestral generations*. A population can save arbitrary number of ancestral generations. It is possible to directly access an ancestor, or make an ancestral generation the current generation for more efficient access.
- Insert or remove loci, resize (shrink or expand) a population, sample from a population, or merge with other populations.
- Manipulate population variables and evaluate expressions in this *local namespace*.
- Save and load a population.

class `population` (*size=[], ploidy=2, loci=[], chromTypes=[], lociPos=[], ancestralGens=0, chromNames=[], alleleNames=[], lociNames=[], infoFields=[]*)

The following parameters are used to create a population object:

size: A list of subpopulation sizes. The length of this list determines the number of subpopulations of this population. If there is no subpopulation, *size*=`[popSize]` can be written as *size*=`popSize`.

ploidy: Number of homologous sets of chromosomes. Default to `2` (diploid). For efficiency considerations, all chromosomes have the same number of homologous sets, even if some chromosomes (e.g. mitochondrial) or some individuals (e.g. males in a haplodiploid population) have different numbers of homologous sets. The first case is handled by setting *chromTypes* of each chromosome. Only the haplodiploid populations are handled for the second case, for which `ploidy=Haplodiploid` should be used.

loci: A list of numbers of loci on each chromosome. The length of this parameter determines the number of chromosomes. Default to `[1]`, meaning one chromosome with a single locus.

chromTypes: A list that specifies the type of each chromosome, which can be `Autosome`, `ChromosomeX`, `ChromosomeY`, or `Mitochondrial`. All chromosomes are assumed to be autosomes if this parameter is ignored. Sex chromosome can only be specified in a diploid population where the sex of an individual is determined by the existence of these chromosomes using the `XX` (`Female`) and `XY` (`Male`) convention. Both sex chromosomes have to be available and be specified only once. Because chromosomes `X` and `Y` are treated as two chromosomes, recombination on the pseudo-autosomal regions of the sex chromsomes is not supported. A `Mitochondrial` chromosome only exists in females and is inherited maternally.

lociPos: Positions of all loci on all chromosome, as a list of float numbers. Default to `1`, `2`, ... etc on each chromosome. Positions on the same chromosome should be ordered. A nested list that specifies positions of loci on each chromosome is also acceptable.

ancestralGens: Number of the most recent ancestral generations to keep during evolution. Default to `0`, which means only the current generation will be kept. If it is set to `-1`, all ancestral generations will be kept in this population (and exhaust your computer RAM quickly).

chromNames: A list of chromosome names. Default to `chrom1`, `chrom2`, ... etc.

alleleNames: A list of allele names for all markers. For example, *alleleNames*=`('A','C','T','G')` names allele `0 - 3'A'`, `'C'`, `'T'`, and `'G'` respectively. Note that simuPOP does not yet support locus-specific allele names.

lociNames: A list or a matrix (separated by chromosomes) of names for each locus. Default to `"locX-Y"` where `X` and `Y` are 1-based chromosome and locus indexes, respectively.

infoFields: Names of information fields (named float number) that will be attached to each individual.

ancestor(*ind, gen*)
Refrence to an individual `ind` in an ancestral generation

ancestor(*ind, subPop, gen*)
Refrence to an individual `ind` in a specified subpopulaton or an ancestral generation

clone(*keepAncestralPops=-1*)
Copy a population, with the option to keep all (default), no, or a given number of ancestral generations (*keepAncestralPops* = `-1`, `0`, or a positive number, respectively). Note that Python statement `pop1 = pop` only creates a reference to an existing population `pop`.

save(*filename*)
Save population to a file *filename*. The population can be restored from this file, using a global function `LoadPopulation(filename)`.

absIndIndex(*idx, subPop*)
Return the absolute index of an individual *idx* in subpopulation *subPop*.

numSubPop()
Return the number of subpopulations in a population. Return 1 if there is no subpopulation structure.

popSize()
Return the total number of individuals in all subpopulations.

subPopBegin(*subPop*)
Return the index of the first individual in subpopulation *subPop*. An `IndexError` will be raised if *subPop* is out of range.

subPopEnd(*subPop*)
Return the index of the last individual in subpopulation *subPop* plus `1`, so that `range(subPopBegin(subPop), subPopEnd(subPop)` can iterate through the index of all individuals in subpopulation *subPop*.

subPopIndPair(*idx*)
Return the subpopulation ID and relative index of an individual, given its absolute index `idx`.

subPopSize(*subPop*)
Return the size of a subpopulation (`subPopSize(sp)`) or a virtual subpopulation (`subPopSize([sp, vsp])`).

subPopSizes()
Return the sizes of all subpopulations in a list. Virtual subpopulations are not considered.

numVirtualSubPop()
Return the number of virtual subpopulations (VSP) defined by a VSP splitter. Return `0` if no VSP is defined.

setVirtualSplitter(*splitter*)
Set a VSP *splitter* to the population, which defines the same VSPs for all subpopulations. If different VSPs are needed for different subpopulations, a `combinedSplitter` can be used to make these VSPs available to all subpopulations.

virtualSubPopName(*subPop*)
Return the name of a virtual subpopulation *subPop* (specified by a `(sp, vsp)` pair). Because VSP names are the same across all subpopulations, a single VSP index is also acceptable.

individual(*idx, subPop=0*)
Return a refernce to individual *ind* in subpopulation *subPop*.

individuals()
Return a Python iterator that can be used to iterate through all individuals in a population.

individuals(*subPop*)
Return an iterator that can be used to iterate through all individuals in a subpopulation (`subPop=spID`) or a virtual subpopulation (`subPop=[spID, vspID]`).

genotype()
Return an editable array of the genotype of all individuals in this population.

genotype(*subPop*)
Return an editable array of the genotype of all individuals in subpopulation *subPop*.

setGenotype(*geno*)
Fill the genotype of all individuals of a population using a list of alleles *geno*. *geno* will be reused if its length is less than `popSize()*totNumLoci()*ploidy()`.

setGenotype(*geno, subPop*)
Fill the genotype of all individuals of in subpopulation *subPop* using a list of alleles *geno*. *geno* will be reused if its length is less than `subPopSize(subPop)*totNumLoci()*ploidy()`.

ancestor(*idx, gen*)
Return a reference to individual `idx` in ancestral generation `gen`. The correct individual will be returned even if the current generation is not the present one (see `useAncestralGen`).

ancestor(*ind, subPop, gen*)
Return a reference to individual `idx` of subpopulation *subPop* in ancestral generation `gen`.

ancestralGens()
Return the actual number of ancestral generations stored in a population, which does not necessarily equal to the number set by `setAncestralDepth()`.

setAncestralDepth(*depth*)
Set the intended ancestral depth of a population to *depth*, which can be `0` (does not store any ancestral generation), `-1` (store all ancestral generations), and a positive number (store *depth* ancestral generations.

useAncestralGen(*idx*)
Making ancestral generation *idx* (`0` for current generation, `1` for parental generation, `2` for grand-parental generation, etc) the current generation. This is an efficient way to access population properties of an ancestral generation. `useAncestralGen(0)` should always be called to restore the correct order of ancestral generations.

addChrom(*lociPos, lociNames=[], chromName="", chromType=Autosome*)
Add chromosome *chromName* with given type *chromType* to a population, with loci *lociNames* inserted at position *lociPos*. *lociPos* should be ordered. *lociNames* and *chromName* should not exist in the current population. If they are not specified, simuPOP will try to assign default names, and raise a `ValueError` if the default names have been used.

addChromFromPop(*pop*)

Add chromosomes in population *pop* to the current population. Population *pop* should have the same number of individuals as the current population in the current and all ancestral generations. This function merges genotypes on the new chromosomes from population `pop` individual by individual.

addIndFromPop(*pop*)

Add all individuals, including ancestors, in *pop* to the current population. Two populations should have the same genotypic structures and number of ancestral generations. Subpopulations in population *pop* are kept.

addLoci(*chrom, pos, names=[]*)

Insert loci *names* at positions *pos* on chromosome *chrom*. These parameters should be lists of the same length, although *names* may be ignored, in which case random names will be given. Alleles at inserted loci are initialized with zero alleles. Note that loci have to be added to existing chromosomes. If loci on a new chromosome need to be added, function `addChrom` should be used. This function returns indexes of the inserted loci.

addLociFromPop(*pop*)

Add loci from population *pop*, chromosome by chromosome. Added loci will be inserted according to their position. Their position and names should not overlap with any locus in the current population. Population *pop* should have the same number of individuals as the current population in the current and all ancestral generations.

mergeSubPops(*subPops=[]*)

Merge subpopulations *subPops*. If *subPops* is empty (default), all subpopulations will be merged. Subpopulations *subPops* do not have to be adjacent to each other. The ID of the first subpopulation in parameter *subPops* will become the ID of the new large subpopulation. Other subpopulations will keep their IDs although their sizes become zero. Function `removeEmptySubPops` can be used to remove these empty subpopulation.

removeEmptySubPops()

Remove empty subpopulations by adjusting subpopulation IDs.

removeIndividuals(*inds*)

Remove individuals *inds* (absolute indexes) from the current population. A subpopulation will be kept even if all individuals from it are removed.

removeLoci(*loci=[], keep=[]*)

Remove *loci* (absolute indexes) and genotypes at these loci from the current population. Alternatively, a parameter *keep* can be used to specify loci that will not be removed.

removeSubPops(*subPops*)

Remove all individuals from subpopulations *subPops*. The removed subpopulations will have size zero, and can be removed by function `removeEmptySubPops`.

resize(*newSubPopSizes, propagate=False*)

Resize population by giving new subpopulation sizes *newSubPopSizes*. Individuals at the end of some subpopulations will be removed if the new subpopulation size is smaller than the old one. New individuals will be appended to a subpopulation if the new size is larger. Their genotypes will be set to zero (default), or be copied from existing individuals if *propagate* is set to `True`. More specifically, if a subpopulation with `3` individuals is expanded to `7`, the added individuals will copy genotypes from individual `1`, `2`, `3`, and `1` respectively. Note that this function only resizes the current generation.

splitSubPop(*subPop, sizes, keepOrder=True*)

Split subpopulation *subPop* into subpopulations of given *sizes*, which should add up to the size of subpopulation *subPop*. Alternatively, *sizes* can be a list of proportions (add up to `1`) from which the sizes of new subpopulations are determined. By default, subpopulation indexes will be adjusted so that individuals can keep their original order. That is to say, if subpopulation `1` of a population having four subpopulations is split into three subpopulation, the new subpopulation ID would be `0`, `1.1->1`, `1.2->2`, `1.3->3`, `2->4`, `3->5`. If *keepOrder* is set to `False`, the subpopulation IDs of existing subpopulations will not

be changed so the new subpopulation IDs of the previous example would be `0`, `1.1->1`, `2`, `3`, `1.2->4`, `1.3->5`.

addInfoField (*field, init=0*)
Add an information field *field* to a population and initialize its values to *init*.

addInfoFields (*fields, init=0*)
Add information fields *fields* to a population and initialize their values to *init*. If an information field alreay exists, it will be re-initialized.

indInfo (*idx*)
Return the information field `idx` (an index) of all individuals as a list.

indInfo (*name*)
Return the information field `name` of all individuals as a list.

indInfo (*idx, subPop*)
Return the information field `idx` (an index) of all individuals in (virtual) subpopulation *subPop* as a list.

indInfo (*name, subPop*)
Return the information field `name` of all individuals in (virtual) subpopulation *subPop* as a list.

setIndInfo (*values, idx*)
Set information field `idx` (an index) of the current population to *values*. *values* will be reused if its length is smaller than `popSize()`.

setIndInfo (*values, name*)
Set information field `name` of the current population to *values*. *values* will be reused if its length is smaller than `popSize()`.

setIndInfo (*values, idx, subPop*)
Set information field `idx` (an index) of a subpopulation (`subPop=sp`) or a virtual subpopulation (`subPop=[sp, vsp]`) to *values*. *values* will be reused if its length is smaller than `subPopSize(subPop)`.

setIndInfo (*values, name, subPop*)
Set information field `name` of a subpopulation (`subPop=sp`) or a virtual subpopulation (`subPop=[sp, vsp]`) to *values*. *values* will be reused if its length is smaller than `subPopSize(subPop)`.

setInfoFields (*fields, init=0*)
Set information fields *fields* to a population and initialize them with value *init*. All existing information fields will be removed.

dvars (*subPop=-1*)
Return a wrapper of Python dictionary returned by `vars(subPop)` so that dictionary keys can be accessed as attributes. For example `pop.dvars().alleleFreq` is equivalent to `pop.vars()["alleleFreq"]`.

vars (*subPop=-1*)
Return variables of a population. If `subPop` is given, return a dictionary for specified subpopulation.

1.2 Virtual subpopulation splitters

1.2.1 Class `vspSplitter`

This class is the base class of all virtual subpopulation (VSP) splitters, which provide ways to define groups of individuals in a subpopulation who share certain properties. A splitter defines a fixed number of named VSPs. They do not have to add up to the whole subpopulation, nor do they have to be distinct. After a splitter is assigned to a population, many functions and operators can be applied to individuals within specified VSPs. Only one VSP splitter

can be assigned to a population, which defined VSPs for all its subpopulations. It different splitters are needed for different subpopulations, a `combinedSplitter` should be.

class `vspSplitter`()
This is a virtual class that cannot be instantiated.

`clone`()
All VSP splitter defines a `clone()` function to create an identical copy of itself.

`name`(*vsp*)
Return the name of VSP *vsp* (an index between `0` and `numVirtualSubPop()`).

`numVirtualSubPop`()
Return the number of VSPs defined by this splitter.

1.2.2 Class `sexSplitter`

This splitter defines two VSPs by individual sex. The first VSP consists of all male individuals and the second VSP consists of all females in a subpopulation.

class `sexSplitter`()
Create a sex splitter that defines male and female VSPs.

`name`(*vsp*)
Return `"Male"` if *vsp=0* and `"Female"` otherwise.

`numVirtualSubPop`()
Return `2`.

1.2.3 Class `affectionSplitter`

This class defines two VSPs according individual affection status. The first VSP consists of unaffected invidiauls and the second VSP consists of affected ones.

class `affectionSplitter`()
Create a splitter that defined two VSPs by affection status.

`name`(*vsp*)
Return `"Unaffected"` if *vsp=0* and `"Affected"` if *vsp=1*.

`numVirtualSubPop`()
Return 2.

1.2.4 Class `infoSplitter`

This splitter defines VSPs according to the value of an information field of each indivdiual. A VSP is defined either by a value or a range of values.

class `infoSplitter`(*field, values=[], cutoff=[]*)
Create an infomration splitter using information field *field*. If parameter *values* is specified, each item in this list defines a VSP in which all individuals have this value at information field *field*. If a set of cutoff values are defined in parameter *cutoff*, individuals are grouped by intervals defined by these cutoff values. For example, `cutoff=[1,2]` defines three VSPs with `v < 1`, `1 <= v < 2` and `v >=2` where `v` is the value of an individual at information field *field*. Of course, only one of the parameters *values* and *cutoff* should be defined, values in *cutoff* should be distinct, and in an increasing order.

name(*vsp*)
Return the name of a VSP *vsp*, which is `field = value` if VSPs are defined by values in parameter *values*, or `field < value` (the first VSP), `v1 <= field < v2` and `field >= v` (the last VSP) if VSPs are defined by cutoff values.

numVirtualSubPop()
Return the number of VSPs defined by this splitter, which is the length parameter *values* or the length of *cutoff* plus one, depending on which parameter is specified.

1.2.5 Class `proportionSplitter`

This splitter divides subpopulations into several VSPs by proportion.

class proportionSplitter(*proportions=[]*)
Create a splitter that divides subpopulations by *proportions*, which should be a list of float numbers (between `0` and `1`) that add up to `1`.

name(*vsp*)
Return the name of VSP *vsp*, which is `"Prop p"` where `p=propotions[vsp]`.

numVirtualSubPop()
Return the number of VSPs defined by this splitter, which is the length of parameter *proportions*.

1.2.6 Class `rangeSplitter`

This class defines a splitter that groups individuals in certain ranges into VSPs.

class rangeSplitter(*ranges*)
Create a splitter according to a number of individual ranges defined in *ranges*. For example, `rangeSplitter(ranges=[[0, 20], [40, 50]])` defines two VSPs. The first VSP consists of individuals `0`, `1`, ..., `19`, and the sceond VSP consists of individuals `40`, `41`, ..., `49`. Note that a nested list has to be used even if only one range is defined.

name(*vsp*)
Return the name of VSP *vsp*, which is `"Range [a, b]"` where `[a, b]` is range `ranges[vsp]`.

numVirtualSubPop()
Return the number of VSPs, which is the number of ranges defined in parameter *ranges*.

1.2.7 Class `genotypeSplitter`

This class defines a VSP splitter that defines VSPs according to individual genotype at specified loci.

class genotypeSplitter(*loci (or locus), alleles, phase=False*)
Create a splitter that defined VSPs by individual genotype at loci *loci* (or *locus* if only one locus is used). Each list in a list *allele* defines a VSP, which is a list of allowed alleles at these *loci*. If only one VSP is defined, the outer list of the nested list can be ignored. If phase if true, the order of alleles in each list is significant. If more than one set of alleles are given, individuals having either of them is qualified.

For example, in a haploid population, `locus=1, alleles=[0, 1]` defines a VSP with individuals having allele `0` or `1` at locus `1`, `alleles=[[0, 1], [2]]` defines two VSPs with indivdiuals in the second VSP having allele `2` at locus `1`. If multiple loci are involved, alleles at each locus need to be defined. For example, VSP defined by `loci=[0, 1], alleles=[0, 1, 1, 1]` consists of individuals having alleles `[0, 1]` or `[1, 1]` at loci `[0, 1]`.

In a haploid population, `locus=1, alleles=[0, 1]` defines a VSP with individuals having genotype `[0, 1]` or `[1, 0]` at locus `1`. `alleles[[0, 1], [2, 2]]` defines two VSPs with indivdiuals in the second VSP having genotype `[2, 2]` at locus `1`. If *phase* is set to `True`, the first VSP will only has individuals with genotype `[0, 1]`. In the multiple loci case, alleles should be arranged by haplotypes, for example, `loci=[0, 1], alleles=[0, 0, 1, 1], phase=True` defines a VSP with individuals having genotype `-0-0-`, `-1-1-` at loci `0` and `1`. If `phase=False` (default), genotypes `-1-1-`, `-0-0-`, `-0-1-` and `-1-0-` are all allowed.

name(*vsp*)
Return name of VSP *vsp*, which is `"Genotype loc1,loc2:genotype"` as defined by parameters *loci* and *alleles*.

numVirtualSubPop()
Number of virtual subpops of subpopulation sp

1.2.8 Class `combinedSplitter`

This splitter takes several splitters and stacks their VSPs together. For example, if the first splitter defines `3` VSPs and the second splitter defines `2`, the two VSPs from the second splitter becomes the fourth (index `3`) and the fifth (index `4`) VSPs of the combined splitter. This splitter is usually used to define different types of VSPs to a population.

class combinedSplitter(*splitters=[]*)
Create a combined splitter using a list of *splitters*. For example, `combinedSplitter([sexSplitter(), affectionSplitter()])` defines a combined splitter with four VSPs.

name(*vsp*)
Return the name of a VSP *vsp*, which is the name a VSP defined by one of the combined splitters.

numVirtualSubPop()
Return the number of VSPs defined by this splitter, which is the sum of the number of VSPs of all combined splitters.

1.3 Mating Scheme

1.3.1 Class `mating`

The base class of all mating schemes - a required parameter of `simulator`Mating schemes specify how to generate offspring from the current population. It must be provided when a simulator is created. Mating can perform the following tasks:

- change population/subpopulation sizes;
- randomly select parent(s) to generate offspring to populate the offspring generation;
- apply *during-mating* operators;
- apply selection if applicable.

class mating(*newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None, subPop=[], weight=0*)
Create a mating scheme (do not use this base mating scheme, use one of its derived classes instead) By default, a mating scheme keeps a constant population size, generates one offspring per mating event. These can be changed using certain parameters. `newSubPopSize`, `newSubPopSizeExpr` and `newSubPopSizeFunc` can be used to specify subpopulation sizes of the offspring generation.

newSubPopSize: An array of subpopulations sizes, should have the same number of subpopulations as the current population

newSubPopSizeExpr: An expression that will be evaluated as an array of new subpopulation sizes

newSubPopSizeFunc: A function that takes parameters `gen` (generation number) and `oldsize` (an array of current population size) and return an array of subpopulation sizes of the next generation. This is usually easier to use than its expression version of this parameter.

subPop: If this parameter is given, the mating scheme will be applied only to the given (virtual) subpopulation. This is only used in heteroMating where mating schemes are passed to.

weight: When subPop is virtual, this is used to detemine the number of offspring for this mating scheme. Weight can be

- 0 (default) the weight will be proportional to the current (virtual) subpopulation size. If other virutal subpopulation has non-zero weight, this virtual subpopulation will produce no offspring (weight 0).
- any negative number -n: the size will be n*m where m is the size of the (virtual) subpopulation of the parental generation.
- any positive number n: the size will be determined by weights from all (virtual) subpopulations.

clone()
Deep copy of a mating scheme

submitScratch(*pop, scratch*)
A common submit procedure is defined.

1.3.2 Class `noMating` (Applicable to `all ploidy`)

A mating scheme that does nothing In this scheme, there is

- no mating. Parent generation will be considered as offspring generation.
- no subpopulation change. *During-mating* operators will be applied, but the return values are not checked. I.e., subpopulation size parameters will be ignored although some during-mating operators might be applied.

Note that because the offspring population is the same as parental population, this mating scheme can not be used with other mating schemes in a heterogeneous mating scheme. cloneMating is recommended for that purpose.

class noMating(*numOffspring=1.0, numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None, subPop=[], weight=0*)
Creat a scheme with no mating

Note All parameters are ignored!

clone()
Deep copy of a scheme with no mating

1.3.3 Class `cloneMating` (Applicable to `all ploidy`)

A clone mating that copy everyone from parental to offspring generation. Note that

- selection is not considered (fitness is ignored)
- sequentialParentMating is used. If offspring (virtual) subpopulation size is smaller than parental subpopulation size, not all parents will be cloned. If offspring (virtual) subpopulation size is larger, some parents will be cloned more than once.

- numOffspring interface is respected.
- during mating operators are applied.

class cloneMating(*numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None, subPop=[], weight=0*)

Create a binomial selection mating scheme Please refer to class `mating` for parameter descriptions.

clone()

Deep copy of a binomial selection mating scheme

1.3.4 Class `binomialSelection` (Applicable to `all ploidy`)

A mating scheme that uses binomial selection, regardless of sex No sex information is involved (binomial random selection). Offspring is chosen from parental generation by random or according to the fitness values. In this mating scheme,

- `numOffspring` protocol is honored;
- population size changes are allowed;
- selection is possible;
- haploid population is allowed.

class binomialSelection(*numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None, subPop=[], weight=0*)

Create a binomial selection mating scheme Please refer to class `mating` for parameter descriptions.

clone()

Deep copy of a binomial selection mating scheme

1.3.5 Class `baseRandomMating` (Applicable to `diploid only`)

This base class defines a general random mating scheme that makes full use of a general random parents chooser, and a Mendelian offspring generator. A general random parents chooser allows selection without replacement, polygemous parents selection (a parent with more than one partners), and the definition of several alpha individuals. Direct use of this mating scheme is not recommended. `randomMating`, `monogemousMating`, `polygemousMating`, `alphaMating` are all special cases of this mating scheme. They should be used whenever possible.

class baseRandomMating(*replacement=True, replenish=False, polySex=Male, polyNum=1, alphaSex=Male, alphaNum=0, alphaField=string, numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None, contWhenUniSex=True, subPop=[], weight=0*)

FIXME: No document

replacement: If set to `True`, a parent can be chosen to mate again. Default to False.

replenish: In case that `replacement=True`, whether or not replenish a sex group when it is exhausted.

polySex: Sex of polygamous mating. Male for polygyny, Female for polyandry.

polyNum: Number of sex partners.

alphaSex: The sex of the alpha individual, i.e. alpha male or alpha female who be the only mating individuals in their sex group.

alphaNum: Number of alpha individuals. If `infoField` is not given, `alphaNum` random individuals with `alphaSex` will be chosen. If selection is enabled, individuals with higher+ fitness values have higher probability to be selected. There is by default no alpha individual (`alphaNum` = 0).

alphaField: If an information field is given, individuals with non-zero values at this information field are alpha individuals. Note that these individuals must have `alphaSex`.

clone()
Deep copy of a random mating scheme

1.3.6 Class `randomMating` (Applicable to `diploid only`)

A mating scheme of basic sexually random mating In this scheme, sex information is considered for each individual, and ploidy is always 2. Within each subpopulation, males and females are randomly chosen. Then randomly get one copy of chromosomes from father and mother. If only one sex exists in a subpopulation, a parameter (`contWhenUniSex`) can be set to determine the behavior. Default to continuing without warning.

class randomMating(*numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="", contWhenUniSex=True, subPop=[], weight=0*)
Please refer to class `mating` for descriptions of other parameters.

contWhenUniSex: Continue when there is only one sex in the population. Default to `True`.

clone()
Deep copy of a random mating scheme

1.3.7 Class `selfMating` (Applicable to `diploid only`)

A mating scheme of selfing In this mating scheme, a parent is choosen randomly, acts both as father and mother in the usual random mating. The parent is chosen randomly, regardless of sex. If selection is turned on, the probability that an individual is chosen is proportional to his/her fitness.

class selfMating(*numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="", contWhenUniSex=True, subPop=[], weight=0*)
Create a self mating scheme Please refer to class `mating` for descriptions of other parameters.

contWhenUniSex: Continue when there is only one sex in the population. Default to `True`.

clone()
Deep copy of a self mating scheme

1.3.8 Class `monogamousMating` (Applicable to `diploid only`)

A mating scheme of monogamy This mating scheme is identical to random mating except that parents are chosen without replacement. Under this mating scheme, offspring share the same mother must share the same father. In case that all parental pairs are exhausted, parameter `replenish=True` allows for the replenishment of one or both sex groups.

class monogamousMating(*replenish=False, numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="", contWhenUniSex=True, subPop=[], weight=0*)

REPLENISH This parameter allows replenishment of one or both parental sex groups in case that they are are exhausted. Default to False. Please refer to class mating for descriptions of other parameters.

clone()

Deep copy of a random mating scheme

1.3.9 Class polygamousMating (Applicable to diploid only)

A mating scheme of polygymy or polyandry This mating scheme is composed of a random parents chooser that allows for polygamous mating, and a mendelian offspring generator. In this mating scheme, a male (or female) parent will have more than one sex partner (numPartner). Parents returned from this parents chooser will yield the same male (or female) parents, each with varying partners.

class polygamousMating(*polySex=Male, polyNum=1, replacement=False, replenish=False, numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="", contWhenUniSex=True, subPop=[], weight=0*)

FIXME: No document

polySex: Sex of polygamous mating. Male for polygyny, Female for polyandry.

polyNum: Number of sex partners.

replacement: If set to True, a parent can be chosen to mate again. Default to False.

replenish: In case that replacement=True, whether or not replenish a sex group when it is exhausted. Please refer to class mating for descriptions of other parameters.

clone()

Deep copy of a random mating scheme

1.3.10 Class consanguineousMating (Applicable to diploid only)

A mating scheme of consanguineous mating In this mating scheme, a parent is choosen randomly and mate with a relative that has been located and written to a number of information fields.

class consanguineousMating(*relativeFields=[], func=None, param=None, replacement=False, replenish=True, numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="", contWhenUniSex=True, subPop=[], weight=0*)

Create a consanguineous mating scheme This mating scheme randomly choose a parent and then choose his/her spouse from indexes stored in infoFields.

Please refer to infoParentsChooser and mendelianOffspringGenerator for other parameters.

relativeFields: The information fields that stores indexes to other individuals in a population. If more than one valid (positive value) indexes exist, a random index will be chosen. (c.f. infoParentsChooser) If there is no individual having any valid index, the second parent will be chosen randomly from the whole population.

func: A python function that can be used to prepare the indexes of these information fields. For example, functions population::locateRelatives and/or population::setIndexesOfRelatives can be used to locate certain types of relatives of each individual.

param: An optional parameter that can be passed to func.

clone()
Deep copy of a consanguineous mating scheme

1.3.11 Class `alphaMating` (Applicable to `diploid only`)

Only a number of alpha individuals can mate with individuals of opposite sex. This mating scheme is composed of an random parents chooser with alpha individuals, and a Mendelian offspring generator. That is to say, a certain number of alpha individual (male or female) are determined by `alphaNum` or an information field. Then, only these alpha individuals are able to mate with random individuals of opposite sex.

class alphaMating(*alphaSex=Male, alphaNum=0, alphaField=string, numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="", subPop=[], weight=0*)

Please refer to class `mating` for descriptions of other parameters. Note: If selection is enabled, it works regularly on on-alpha sex, but works twice on alpha sex. That is to say, `alphaNum` alpha indiviudals are chosen selectively, and selected again during mating.

alphaSex: The sex of the alpha individual, i.e. alpha male or alpha female who be the only mating individuals in their sex group.

alphaNum: Number of alpha individuals. If `infoField` is not given, `alphaNum` random individuals with `alphaSex` will be chosen. If selection is enabled, individuals with higher+ fitness values have higher probability to be selected. There is by default no alpha individual (`alphaNum` = 0).

alphaField: If an information field is given, individuals with non-zero values at this information field are alpha individuals. Note that these individuals must have `alphaSex`.

clone()
Deep copy of a random mating scheme

1.3.12 Class `haplodiploidMating` (Applicable to `haplodiploid only`)

Haplodiploid mating scheme of many hymemopterans This mating scheme is composed of an alphaParentChooser and a haplodiploidOffspringGenerator. The alphaParentChooser chooses a single Female randomly or from a given information field. This female will mate with random males from the colony. The offspring will have one of the two copies of chromosomes from the female parent, and the first copy of chromosomes from the male parent. Note that if a recombinator is used, it should disable recombination of male parent.

class haplodiploidMating(*alphaSex=Female, alphaNum=1, alphaField=string, numOffspring=1., numOffspringFunc=None, maxNumOffspring=0, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex, newSubPopSize=[], newSubPopSizeFunc=None, newSubPopSizeExpr="", subPop=[], weight=0*)

Please refer to class `mating` for descriptions of other parameters.

alphaSex: Sex of the alpha individual. Default to Female.

alphaNum: Number of alpha individual. Default to one.

alphaField: Information field that identifies the queen of the colony. By default, a random female will be chosen.

clone()
Deep copy of a random mating scheme

1.3.13 Class `pyMating` (Applicable to `all ploidy`)

A Python mating scheme This hybrid mating scheme does not have to involve a python function. It requires a parent

chooser, and an offspring generator. The parent chooser chooses parent(s) and pass them to the offspring generator to produce offspring.

class pyMating(*chooser, generator, newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None, subPop=[], weight=0*)
Create a Python mating scheme

chooser: A parent chooser that chooses parent(s) from the parental generation.

generator: An offspring generator that produce offspring of given parents.

clone()
Deep copy of a Python mating scheme

1.3.14 Class `heteroMating` (Applicable to `diploid only`)

A heterogeneous mating scheme that applies a list of mating schemes to different (virtual) subpopulations.

class heteroMating(*matingSchemes, newSubPopSize=[], newSubPopSizeExpr="", newSubPopSizeFunc=None, shuffleOffspring=True, subPop=[], weight=0*)
Create a heterogeneous Python mating scheme Parameter subpop, virtualSubPOp and weight of this mating scheme is ignored.

matingSchemes: A list of mating schemes. If parameter `subPop` of an mating scheme is specified, it will be applied to specific subpopulation. If `virtualSubPop` if specified, it will be applied to specifc virtual subpopulations.

clone()
Deep copy of a Python mating scheme

1.3.15 Class `sequentialParentChooser` (Applicable to `all ploidy`)

This parent chooser chooses a parent linearly, regardless of sex or fitness values (selection is not considered).

class sequentialParentChooser()
FIXME: No document

clone()
FIXME: No document

1.3.16 Class `sequentialParentsChooser` (Applicable to `all ploidy`)

This parents chooser chooses two parents sequentially. The parents are chosen from their respective sex groups. Selection is not considered.

class sequentialParentsChooser()
FIXME: No document

clone()
FIXME: No document

1.3.17 Class `randomParentChooser` (Applicable to `all ploidy`)

This parent chooser chooses a parent randomly from the parental generation. If selection is turned on, parents are chosen with probabilities that are proportional to their fitness values. Sex is not considered. Parameter

`replacement` determines if a parent can be chosen multiple times. In case that `replacement=false`, paremeter `replenish=true` allows restart of the process if all parents are exhausted. Note that selection is not allowed when `replacement=false` because this poses a particular order on individuals in the offspring generation.

class randomParentChooser(*replacement=True, replenish=False*)

FIXME: No document

replacement: If replacement is false, a parent can not be chosen more than once.

replenish: If all parent has been chosen, choose from the whole parental population again.

clone()

FIXME: No document

1.3.18 Class `randomParentsChooser` (Applicable to `all ploidy`)

This parent chooser chooses two parents randomly, a male and a female, from their respective sex groups randomly. If selection is turned on, parents are chosen from their sex groups with probabilities that are proportional to their fitness values. If parameter `replacement` is false, a chosen pair of parents can no longer be selected. This feature can be used to simulate monopoly. If `replenish` is true, a sex group can be replenished when it is exhausted. Note that selection is not allowed in the case of monopoly because this poses a particular order on individuals in the offspring generation. This parents chooser also allows polygamous mating by reusing a parent multiple times when returning parents, and allows specification of a few alpha individuals who will be the only mating individuals in their sex group.

class randomParentsChooser(*replacement=True, replenish=False, polySex=Male, polyNum=1, alphaSex=Male, alphaNum=0, alphaField=string*)

Note: If selection is enabled, it works regularly on on-alpha sex, but works twice on alpha sex. That is to say, `alphaNum` alpha indiviudals are chosen selectively, and selected again during mating.

replacement: Choose with (`True`, default) or without (`False`) replacement. When choosing without replacement, parents will be paired and can only mate once.

replenish: If set to true, one or both sex groups will be replenished if they are exhausted.

polySex: Male (polygyny) or Female (polyandry) parent that will have `polyNum` sex partners.

polyNum: Number of sex partners.

alphaSex: The sex of the alpha individual, i.e. alpha male or alpha female who be the only mating individuals in their sex group.

alphaNum: Number of alpha individuals. If `infoField` is not given, `alphaNum` random individuals with `alphaSex` will be chosen. If selection is enabled, individuals with higher fitness values have higher probability to be selected. There is by default no alpha individual (`alphaNum` = 0).

alphaField: If an information field is given, individuals with non-zero values at this information field are alpha individuals. Note that these individuals must have `alphaSex`.

clone()

FIXME: No document

1.3.19 Class `infoParentsChooser` (Applicable to `all ploidy`)

This parents chooser choose an individual randomly, but choose his/her spouse from a given set of information fields, which stores indexes of individuals in the same generation. A field will be ignored if its value is negative, or if sex is compatible. Depending on what indexes are stored in these information fields, this parent chooser can be used to implement consanguineous mating where close relatives are located for each individual, or certain non-random mating schemes where each individual can only mate with a small number of pre-determinable individuals. This parent

chooser (currently) uses `randomParentChooser` to choose one parent and randomly choose another one from the information fields. Because of potentially non-even distribution of valid information fields, the overall process may not be as random as expected, especially when selection is applied. Note: if there is no valid individual, this parents chooser works like a double parentChooser.

class `infoParentsChooser`(*infoFields=[], replacement=True, replenish=False*)

FIXME: No document

infoFields: Information fields that store index of matable individuals.

replacement: If replacement is false, a parent can not be chosen more than once.

replenish: If all parent has been chosen, choose from the whole parental population again.

`clone`()

FIXME: No document

1.3.20 Class `pyParentsChooser` (Applicable to `all ploidy`)

This parents chooser accept a Python generator function that yields repeatedly an index (relative to each subpopulation) of a parent, or indexes of two parents as a Python list of tuple. The generator function is responsible for handling sex or selection if needed.

class `pyParentsChooser`(*parentsGenerator*)

FIXME: No document

parentsGenerator: A Python generator function

`clone`()

FIXME: No document

`finalize`(*pop, sp*)

FIXME: No document

1.3.21 Class `cloneOffspringGenerator` (Applicable to `all ploidy`)

Clone offspring generator copies parental geneotype to a number of offspring. Only one parent is accepted. The number of offspring produced is controled by parameters `numOffspring`, `numOffspringFunc`, `maxNumOffspring` and `mode`. Parameters `sexParam` and `sexMode` is ignored.

class `cloneOffspringGenerator`(*numOffspring=1, numOffspringFunc=None, maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex*)

FIXME: No document

sexParam: Ignored because sex is copied from the parent.

sexMode: Ignored because sex is copied from the parent.

`clone`()

FIXME: No document

1.3.22 Class `selfingOffspringGenerator` (Applicable to `diploid only`)

Selfing offspring generator works similarly as a mendelian offspring generator but a single parent produces both the paternal and maternal copy of the offspring chromosomes. This offspring generator accepts a dipload parent. A random copy of the parental chromosomes is chosen randomly to form the parental copy of the offspring chromosome, and is chosen randomly again to form the maternal copy of the offspring chromosome.

class selfingOffspringGenerator(*numOffspring=1, numOffspringFunc=None, maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex*)

FIXME: No document

clone()

FIXME: No document

1.3.23 Class haplodiploidOffspringGenerator (Applicable to haplodiploid only)

Haplodiploid offspring generator mimics sex-determination in honey bees. Given a female (queen) parent and a male parent, the female is considered as diploid with two set of chromosomes, and the male is condiered as haploid. Actually, the first set of male chromosomes are used. During mating, female produce eggs, subject to potential recombination and gene conversion, while male sperm is identical to the parental chromosome. Female offspring has two sets of chromosomes, one from mother and one from father. Male offspring has one set of chromosomes from his mother.

class haplodiploidOffspringGenerator(*numOffspring=1, numOffspringFunc=None, maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex*)

FIXME: No document

clone()

FIXME: No document

copyParentalGenotype(*parent, it, ploidy, count*)

FIXME: No document

1.3.24 Class mendelianOffspringGenerator (Applicable to diploid only)

Mendelian offspring generator accepts two parents and pass their genotype to a number of offspring following Mendelian's law. Basically, one of the paternal chromosomes is chosen randomly to form the paternal copy of the offspring, and one of the maternal chromosome is chosen randomly to form the maternal copy of the offspring. The number of offspring produced is controled by parameters numOffspring, numOffspringFunc, maxNumOffspring and mode. Recombination will not happen unless a during-mating operator recombinator is used.

class mendelianOffspringGenerator(*numOffspring=1, numOffspringFunc=None, maxNumOffspring=1, mode=MATE_NumOffspring, sexParam=0.5, sexMode=MATE_RandomSex*)

FIXME: No document

clone()

FIXME: No document

formOffspringGenotype(*parent, it, ploidy, count*)

Does not set sex if count == -1.

count: Index of offspring, used to set offspring sex

1.4 Simulator

1.4.1 Class `simulator`

Simulator manages several replicates of a population, evolve them using given mating scheme and operators Simulators combine three important components of simuPOP: population, mating scheme and operator together. A simulator is created with an instance of `population`, a replicate number `rep` and a mating scheme. It makes `rep` number of replicates of this population and control the evolutionary process of them.

The most important function of a simulator is `evolve()`. It accepts an array of operators as its parameters, among which, `preOps` and `postOps` will be applied to the populations at the beginning and the end of evolution, respectively, whereas `ops` will be applied at every generation.

A simulator separates operators into *pre-*, *during-*, and *post-mating* operators. During evolution, a simulator first apply all pre-mating operators and then call the `mate()` function of the given mating scheme, which will call during-mating operators during the birth of each offspring. After mating is completed, post-mating operators are applied to the offspring in the order at which they appear in the operator list.

Simulators can evolve a given number of generations (the `end` parameter of `evolve`), or evolve indefinitely until a certain type of operators called terminator terminates it. In this case, one or more terminators will check the status of evolution and determine if the simulation should be stopped. An obvious example of such a terminator is a fixation-checker.

A simulator can be saved to a file in the format of `'txt'`, `'bin'`, or `'xml'`. This allows you to stop a simulator and resume it at another time or on another machine.

class `simulator` (*pop, matingScheme, rep=1*)

Create a simulator

population: A population created by `population()` function. This population will be copied `rep` times to the simulator. Its content will not be changed.

matingScheme: A mating scheme

rep: Number of replicates. Default to `1`.

`addInfoField` (*field, init=0*)

Add an information field to all replicates Add an information field to all replicate, and to the simulator itself. This is important because all populations must have the same genotypic information as the simulator. Adding an information field to one or more of the replicates will compromise the integrity of the simulator.

field: Information field to be added

`addInfoFields` (*fields, init=0*)

Add information fields to all replicates Add given information fields to all replicate, and to the simulator itself.

`clone`()

Deep copy of a simulator

`evolve` (*ops, preOps=[], postOps=[], gen=-1, dryrun=False*)

Evolve all populations *gen* generations, subject to operators *opspreOps* and *postOps*. Operators *preOps* are applied to all populations (subject to applicability restrictions of the operators, imposed by the *rep* parameter of these operators) before evolution. They are usually used to initialize populations. Operators *postOps* are applied to all populations after the evolution.

Operators *ops* are applied during the life cycle of each generation. Depending on the stage of these operators, they can be applied before-, during-, and/or post-mating. These operators can be applied at all or some of the generations, depending the *begin*, *end*, *step*, and *at* parameters of these operators. Populations in a simulator are evolved one by one. At each generation, the applicability of these operators are determined. Pre-mating operators are applied to a population first. A mating scheme is then used to populate

an offspring generation, using applicable during-mating operators. After an offspring generation is successfully generated and becomes the current generation, applicable post-mating operators are applied to it. Because the order at which operators are applied can be important, and the stage(s) at which operators are applied are not always clear, a parameter *dryRun* can be used. If set to `True`, this function will print out the order at which all operators are applied, without actually evolving the populations.

Parameter *gen* can be set to a positive number, which is the number of generations to evolve. If *gen* is negative (default), the evolution will continue indefinitely, until all replicates are stopped by a special kind of operators called *terminators*. At the end of the evolution, the generations that each replicates have evolved are returned.

gen()
Return the current generation number

getPopulation(*rep, destructive=False*)
Return a copy of population `rep`By default return a cloned copy of population `rep` of the simulator. If `destructive==True`, the population is extracted from the simulator, leaving a defunct simulator.

rep: The index number of the replicate which will be obtained

destructive: If true, destroy the copy of population within this simulator. Default to false. `getPopulation(rep, true)` is a more efficient way to get hold of a population when the simulator will no longer be used.

numRep()
Return the number of replicates

population(*rep*)
Return a reference to the `rep` replicate of this simulator.

rep: The index number of replicate which will be accessed

Note: The returned reference is temporary in the sense that the refered population will be invalid after another round of evolution. If you would like to get a persistent population, please use `getPopulation(rep)`.

populations()
Return a iterator that can be used to iterate through all populations in a simulator.

save(*filename*)
Save simulator in `'txt'`, `'bin'` or `'xml'` format

filename: Filename to save the simulator. Default to `simu`.

setAncestralDepth(*depth*)
Set ancestral depth of all replicates

setGen(*gen*)
Set the current generation. Usually used to reset a simulator.

gen: New generation index number

setMatingScheme(*matingScheme*)
Set a new mating scheme

vars(*rep, subPop=-1*)
Return the local namespace of population `rep`, equivalent to `x.population(rep).vars(subPop)`.

dvars(*rep, subPop=-1*)
Return a wrapper of Python dictionary returned by `vars(rep, subPop)` so that dictionary keys can be accessed as attributes. For example `simu.dvars(1).alleleFreq` is equivalent to `simu.vars(1)["alleleFreq"]`.

1.5 Pedigree

1.5.1 Class `pedigree`

FIXME: No document

CHAPTER

TWO

Operator References

This chapter will list all functions, types and operators by category. The reference for `class baseOperator` is in section **??**.

2.1 The common interface of operators

2.1.1 Class `baseOperator`

Base class of all classes that manipulate populations Operators are objects that act on populations. They can be applied to populations directly using their function forms, but they are usually managed and applied by a simulator.

There are three kinds of operators:

- built-in: written in C++, the fastest. They do not interact with Python shell except that some of them set variables that are accessible from Python.

- hybrid: written in C++ but calls a Python function during execution. Less efficient. For example, a hybrid mutator `pyMutator` will go through a population and mutate alleles with given mutation rate. How exactly the allele will be mutated is determined by a user-provided Python function. More specifically, this operator will pass the current allele to a user-provided Python function and take its return value as the mutant allele.

- pure Python: written in Python. The same speed as Python. For example, a `varPlotter` can plot Python variables that are set by other operators. Usually, an individual or a population object is passed to a user-provided Python function. Because arbitrary operations can be performed on the passed object, this operator is very flexible.

Operators can be applied at different stages of the life cycle of a generation. It is possible for an operator to apply multiple times in a life cycle. For example, a `savePopulation` operator might be applied before and after mating to trace parental information. More specifically, operators can be applied at *pre-*, *during-*, *post-mating*, or a combination of these stages. Applicable stages are usually set by default but you can change it by setting `stage=(PreMating|PostMating|DuringMating|PrePostMating|PreDuringMating|DuringPostMating)` parameter. Some operators ignore `stage` parameter because they only work at one stage.

Operators do not have to be applied at all generations. You can specify starting and/or ending generations (parameters `start`, `end`), gaps between applicable generations (parameter `step`), or specific generations (parameter `at`). For example, you might want to start applying migrations after certain burn-in generations, or calculate certain statistics only sparsely. Generation numbers can be counted from the last generation, using negative generation numbers.

Most operators are applied to every replicate of a simulator during evolution. Operators can have outputs, which can be standard (terminal) or a file. Output can vary with replicates and/or generations, and outputs from different operators can be accumulated to the same file to form table-like outputs.

Filenames can have the following format:

- `'filename'` this file will be overwritten each time. If two operators output to the same file, only the last one will succeed;
- `'>filename'` the same as `'filename'`;
- `'>>filename'` the file will be created at the beginning of evolution (`simulator::evolve`) and closed at the end. Outputs from several operators are appended;
- `'>>>filename'` the same as `'>>filename'` except that the file will not be cleared at the beginning of evolution if it is not empty;
- `'>'` standard output (terminal);
- `''` suppress output.

The output filename does not have to be fixed. If parameter `outputExpr` is used (parameter `output` will be ignored), it will be evaluated when a filename is needed. This is useful when you need to write different files for different replicates/generations.

class `baseOperator` (*output, outputExpr, stage, begin, end, step, at, rep, infoFields*)

Common interface for all operators (this base operator does nothing by itself.)

begin: The starting generation. Default to `0`. A negative number is allowed.

end: Stop applying after this generation. A negative numbers is allowed.

step: The number of generations between active generations. Default to `1`.

at: An array of active generations. If given, `stage`, `begin`, `end`, and `step` will be ignored.

rep: Applicable replicates. It can be a valid replicate number, `vectori()` (all replicates, default), or `-1` (only the last replicate). `-1` is useful in adding newlines to a table output.

output: A string of the output filename. Different operators will have different default `output` (most commonly `'>'` or `''`).

outputExpr: An expression that determines the output filename dynamically. This expression will be evaluated against a population's local namespace each time when an output filename is required. For example, `"'>>out%s_%s.xml' % (gen, rep)"` will output to `>>>out1_1.xml` for replicate `1` at generation `1`.

Note

- Negative generation numbers are allowed for parameters `begin`, `end` and `at`. They are interpreted as `endGen + gen + 1`. For example, `begin = -2` in `simu.evolve(..., end=20)` starts at generation `19`.
- `vectori(), -1` are special constant that can only be used in the constructor of an operator. That is to say, explicit test of `rep() == -1` will not work.

`apply`(*pop*)

Apply to one population. It does not check if the operator is activated.

`clone`()

Deep copy of an operator

`diploidOnly`()

Determine if the operator can be applied only for diploid population

haploidOnly()
Determine if the operator can be applied only for haploid population

infoField(*idx*)
Get the information field specified by user (or by default)

infoSize()
Get the length of information fields for this operator

initialize(*pop*)
FIXME: No document

2.2 Initialization

2.2.1 Class `initializer`

Initialize alleles at the start of a generation Initializers are used to initialize populations before evolution. They are set to be `PreMating` operators by default. simuPOP provides three initializers. One assigns alleles by random, one assigns a fixed set of genotypes, and the last one calls a user-defined function.

class initializer(*subPop=[], indRange=[], loci=[], atPloidy=-1, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create an initializer. Default to be always active.

subPop: An array specifies applicable subpopulations

indRange: A `[begin, end]` pair of the range of absolute indexes of individuals, for example, `([1,2])`; or an array of `[begin, end]` pairs, such as `([[1,4],[5,6]])`. This is how you can initialize individuals differently within subpopulations. Note that ranges are in the form of [a,b). I.e., range [4,6] will intialize individual 4, 5, but not 6. As a shortcut for [4,5], you can use [4] to specify one individual.

loci: A vector of locus indexes at which initialization will be done. If empty, apply to all loci.

locus: A shortcut to `loci`

atPloidy: Initialize which copy of chromosomes. Default to all.

clone()
Deep copy of an initializer

2.2.2 Class `initSex` (Function form: `InitSex`)

An operator to initialize individual sex. For convenience, this operator is included by other initializers such as initByFreq, initByValue, or pyInit.

class initSex(*maleFreq=0.5, sex=[], subPop=[], indRange=[], loci=[], atPloidy=-1, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Initialize individual sex.

maleFreq: Male frequency. Default to `0.5`. Sex will be initialized with this parameter.

sex: A list of sexes (Male or Female) and will be applied to individuals in in turn. If specified, parameter `maleFreq` is ignored.

apply(*pop*)
Apply this operator to population `pop`

clone()
Deep copy of an initSex

2.2.3 Class `initByFreq` (Function form: `InitByFreq`)

Initialize genotypes by given allele frequencies, and sex by male frequency This operator assigns alleles at `loci` with given allele frequencies. By default, all individuals will be assigned with random alleles. If `identicalInds=True`, an individual is assigned with random alleles and is then copied to all others. If `subPop` or `indRange` is given, multiple arrays of `alleleFreq` can be given to given different frequencies for different subpopulation or individual ranges.

class `initByFreq`(*alleleFreq=[], identicalInds=False, subPop=[], indRange=[], loci=[], atPloidy=-1, maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=[], infoFields=[]*)

Randomly assign alleles according to given allele frequencies

alleleFreq: An array of allele frequencies. The sum of all frequencies must be 1; or for a matrix of allele frequencies, each row corresponses to a subpopulation or range.

identicalInds: Whether or not make individual genotypes identical in all subpopulations. If `True`, this operator will randomly generate genotype for an individual and spread it to the whole subpopulation in the given range.

sex: An array of sex `[Male, Female, Male...]` for individuals. The length of sex will not be checked. If it is shorter than the number of individuals, sex will be reused from the beginning.

stage: Default to `PreMating`.

`apply`(*pop*)

Apply this operator to population `pop`

`clone`()

Deep copy of the operator `initByFreq`

2.2.4 Class `initByValue` (Function form: `InitByValue`)

Initialize genotype by value and then copy to all individuals Operator `initByValue` gets one copy of chromosomes or the whole genotype (or of those corresponds to `loci`) of an individual and copy them to all or a subset of individuals. This operator assigns given alleles to specified individuals. Every individual will have the same genotype. The parameter combinations should be

- `value - subPop/indRange`: individual in `subPop` or in range(s) will be assigned genotype `value`;
- `subPop/indRange`: `subPop` or `indRange` should have the same length as `value`. Each item of `value` will be assigned to each `subPop` or `indRange`.

class `initByValue`(*value=[], loci=[], atPloidy=-1, subPop=[], indRange=[], proportions=[], maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=[], infoFields=[]*)

Initialize a population by given alleles

value: An array of genotypes of one individual, having the same length as the length of `loci()` or `loci()*ploidy()` or `pop.genoSize()` (whole genotype) or `totNumLoci()` (one copy of chromosomes). This parameter can also be an array of arrays of genotypes of one individual. If `value` is an array of values, it should have the length one, number of subpopulations, or the length of ranges of proportions.

proportions: An array of percentages for each item in `value`. If given, assign given genotypes randomly.

maleFreq: Male frequency

sex: An array of sex `[Male, Female, Male...]` for individuals. The length of sex will not be checked. If length of sex is shorter than the number of individuals, sex will be reused from the beginning.

stages: Default to `PreMating`.

apply(*pop*)
Apply this operator to population `pop`

clone()
Deep copy of the operator `initByValue`

2.2.5 Class `spread` (Function form: `Spread`)

Copy the genotype of an individual to all individuals Function `Spread(ind, subPop)` spreads the genotypes of `ind` to all individuals in an array of subpopulations. The default value of `subPop` is the subpopulation where `ind` resides.

class spread(*ind, subPop=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=[], infoFields=[]*)
Copy genotypes of `ind` to all individuals in `subPop`

apply(*pop*)
Apply this operator to population `pop`

clone()
Deep copy of the operator `spread`

2.2.6 Class `pyInit` (Function form: `PyInit`)

A python operator that uses a user-defined function to initialize individuals. This is a hybrid initializer. Users of this operator must supply a Python function with parameters allele, ploidy and subpopulation indexes `(index, ploidy, subPop)`, and return an allele value. This operator will loop through all individuals in each subpopulation and call this function to initialize populations. The arrange of parameters allows different initialization scheme for each subpopulation.

class pyInit(*func, subPop=[], loci=[], atPloidy=-1, indRange=[], maleFreq=0.5, sex=[], stage=PreMating, begin=0, end=1, step=1, at=[], rep=[], infoFields=[]*)
Initialize populations using given user function

func: A Python function with parameter `(index, ploidy, subPop)`, where

- `index` is the allele index ranging from `0` to `totNumLoci-1`;
- `ploidy` is the index of the copy of chromosomes;
- `subPop` is the subpopulation index.

The return value of this function should be an integer.

loci: A vector of locus indexes. If empty, apply to all loci.

locus: A shortcut to `loci`.

atPloidy: Initialize which copy of chromosomes. Default to all.

stage: Default to `PreMating`.

apply(*pop*)
Apply this operator to population `pop`

clone()
Deep copy of the operator `pyInit`

2.3 Migration

2.3.1 Class `migrator`

Migrate individuals from (virtual) subpopulations to other subpopulations Migrator is the only way to mix genotypes of several subpopulations because mating is strictly within subpopulations in simuPOP. Migrators are quite flexible in simuPOP in the sense that

- migration can happen from and to a subset of subpopulations.
- migration can be done by probability, proportion or by counts. In the case of probability, if the migration rate from subpopulation `a` to `b` is `r`, then everyone in subpopulation `a` will have this probability to migrate to `b`. In the case of proportion, exactly `r*size_of_subPop_a` individuals (chosen by random) will migrate to subpopulation `b`. In the last case, a given number of individuals will migrate.
- new subpopulation can be generated through migration. You simply need to migrate to a subpopulation with a new subpopulation number.

class `migrator`(*rate, mode=MigrByProbability, fromSubPop=[], toSubPop=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a migrator

rate: Migration rate, can be a proportion or counted number. Determined by parameter `mode`. `rate` should be an m by n matrix. If a number is given, the migration rate will be a `m` by `n` matrix of value `r`

mode: One of `MigrByProbability` (default), `MigrByProportion` or `MigrByCounts`

fromSubPop: An array of 'from' subpopulations (a number) or virtual subpopulations (a pair of numbers). Default to all subpopulations. For example, if you define a virtual subpopulation by sex, you can use `fromSubpop=[(0,0), 1]` to choose migrants from the first virtual subpopulation of subpopulation 0, and from subpopulation 1. If a single number `sp` is given, it is intepretted as [sp]. Note that `fromSubPop=(0, 1)` (two subpopulation) is different from `fromSubPop=[(0,1)]` (a virtual subpopulation).

toSubPop: An array of 'to' subpopulations. Default to all subpopulations. If a single subpopulation is specified, `[]` can be ignored.

stage: Default to `PreMating`

Note

- The overall population size will not be changed. (Mating schemes can do that). If you would like to keep the subpopulation sizes after migration, you can use the `newSubPopSize` or `newSubPopSizeExpr` parameter of a mating scheme.
- `rate` is a matrix with dimensions determined by `fromSubPop` and `toSubPop`. By default, `rate` is a matrix with element `r(i,j)`, where `r(i, j)` is the migration rate, probability or count from subpopulation `i` to `j`. If `fromSubPop` and/or `toSubPop` are given, migration will only happen between these subpopulations. An extreme case is 'point migration', `rate=[[r]], fromSubPop=a, toSubPop=b` which migrate from subpopulation `a` to `b` with given rate `r`.

`apply`(*pop*)

Apply the `migrator`

`clone`()

Deep copy of a migrator

`rate`()

Return migration rate

setRates(*rate, mode*)
Set migration rate Format should be `0-0 0-1 0-2, 1-0 1-1 1-2, 2-0, 2-1, 2-2`. For mode `MigrByProbability` or `MigrByProportion`, `0-0,1-1,2-2` will be set automatically regardless of input.

2.3.2 Class `pyMigrator`

A more flexible Python migrator This migrator can be used in two ways

- define a function that accepts a generation number and returns a migration rate matrix. This can be used in various migration rate cases.
- define a function that accepts individuals etc, and returns the new subpopulation ID.

More specifically, `func` can be

- `func(ind)` when neither `loci` nor `param` is given.
- `func(ind, genotype)` when `loci` is given.
- `func(ind, param)` when `param` is given.
- `func(ind, genotype, param)` when both `loci` and `param` are given.

class pyMigrator(*rateFunc=None, indFunc=None, mode=MigrByProbability, fromSubPop=[], toSubPop=[], loci=[], param=None, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a hybrid migrator

rateFunc: A Python function that accepts a generation number, current subpopulation sizes, and returns a migration rate matrix. The migrator then migrate like a usual migrator.

indFunc: A Python function that accepts an individual, optional genotypes and parameters, then returns a subpopulation ID. This method can be used to separate a population according to individual genotype.

stage: Default to `PreMating`

apply(*pop*)
Apply a `pyMigrator`

clone()
Deep copy of a `pyMigrator`

2.3.3 Class `splitSubPop` (Function form: `SplitSubPop`)

Split a subpopulation

class splitSubPop(*which=0, sizes=[], proportions=[], keepOrder=True, randomize=True, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Split a subpopulation Split a subpopulation by sizes or proportions. Individuals are randomly (by default) assigned to the resulting subpopulations. Because mating schemes may introduce certain order to individuals, randomization ensures that split subpopulations have roughly even distribution of genotypes.

which: Which subpopulation to split. If there is no subpopulation structure, use `0` as the first (and only) subpopulation.

sizes: New subpopulation sizes. The sizes should be added up to the original subpopulation (subpopulation `which`) size.

proportions: Proportions of new subpopulations. Should be added up to `1`.

randomize: Whether or not randomize individuals before population split. Default to True.

apply(*pop*)
Apply a `splitSubPop` operator

clone()
Deep copy of a `splitSubPop` operator

2.3.4 Class `mergeSubPops` (Function form: `MergeSubPops`)

Merge subpopulations This operator merges subpopulations `subPops` to a single subpopulation. If `subPops` is ignored, all subpopulations will be merged.

class mergeSubPops(*subPops=[], removeEmptySubPops=False, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Merge subpopulations

subPops: Subpopulations to be merged. Default to all.

apply(*pop*)
Apply a `mergeSubPops` operator

clone()
Deep copy of a `mergeSubPops` operator

2.3.5 Class `resizeSubPops` (Function form: `ResizeSubPops`)

Resize subpopulations This operator resize subpopulations `subPops` to a another size. If `subPops` is ignored, all subpopulations will be resized. If the new size is smaller than the original one, the remaining individuals are discarded. If the new size if greater, individuals will be copied again if propagate is true, and be empty otherwise.

class resizeSubPops(*newSizes=[], subPops=[], propagate=True, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Resize subpopulations

newSizes: Of the specified (or all) subpopulations.

subPops: Subpopulations to be resized. Default to all.

propagate: If true (default) and the new size if greater than the original size, individuals will be copied over.

apply(*pop*)
Apply a `resizeSubPops` operator

clone()
Deep copy of a `resizeSubPops` operator

2.4 Mutation

2.4.1 Class `mutator`

Base class of all mutators. The base class of all functional mutators. It is not supposed to be called directly.

Every mutator can specify `rate` (equal rate or different rates for different loci) and a vector of applicable loci (default to all but should have the same length as `rate` if `rate` has length greater than one).

Maximum allele can be specified as well but more parameters, if needed, should be implemented by individual mutator classes.

There are numbers of possible allelic states. Most theoretical studies assume an infinite number of allelic states to avoid any homoplasy. If it facilitates any analysis, this is however extremely unrealistic.

class mutator(*rate=[], loci=[], maxAllele=0, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a mutator, do not call this constructor directly All mutators have the following common parameters. However, the actual meaning of these parameters may vary according to different models. The only differences between the following mutators are the way they actually mutate an allele, and corresponding input parameters. The number of mutation events at each locus is recorded and can be accessed from the `mutationCount` or `mutationCounts` functions.

rate: Can be a number (uniform rate) or an array of mutation rates (the same length as `loci`)

loci: A vector of locus indexes. Will be ignored only when single rate is specified. Default to all loci.

maxAllele: Maximum allowed allele. Interpreted by each sub mutator class. Default to `pop.maxAllele()`.

apply(*pop*)
Apply a mutator

clone()
Deep copy of a `mutator`

maxAllele()
Return maximum allowable allele number

mutate(*allele*)
Describe how to mutate a single allele

mutationCount(*locus*)
Return mutation count at `locus`

mutationCounts()
Return mutation counts

rate()
Return the mutation rate

setMaxAllele(*maxAllele*)
Set maximum allowable allele

setRate(*rate, loci=[]*)
Set an array of mutation rates

2.4.2 Class `kamMutator` (Function form: `KamMutate`)

K-Allele Model mutator. This mutator mutate an allele to another allelic state with equal probability. The specified mutation rate is actually the 'probability to mutate'. So the mutation rate to any other allelic state is actually $\frac{rate}{K-1}$, where K is specified by parameter `maxAllele`.

class kamMutator(*rate=[], loci=[], maxAllele=0, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a K-Allele Model mutator Please see class `mutator` for the descriptions of other parameters.

rate: Mutation rate. It is the 'probability to mutate'. The actual mutation rate to any of the other `K-1` allelic states are `rate/(K-1)`.

maxAllele: Maximum allele that can be mutated to. For binary libraries, allelic states will be `[0, maxAllele]`. Otherwise, they are `[1, maxAllele]`.

clone()
Deep copy of a `kamMutator`

mutate(*allele*)
Mutate to a state other than current state with equal probability

2.4.3 Class `smmMutator` (Function form: `SmmMutate`)

The stepwise mutation model. The *Stepwise Mutation Model* (SMM) assumes that alleles are represented by integer values and that a mutation either increases or decreases the allele value by one. For variable number tandem repeats(VNTR) loci, the allele value is generally taken as the number of tandem repeats in the DNA sequence.

class smmMutator(*rate=[], loci=[], maxAllele=0, incProb=0.5, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a SMM mutator The SMM is developed for allozymes. It provides better description for these kinds of evolutionary processes.

Please see class `mutator` for the descriptions of other parameters.

incProb: Probability to increase allele state. Default to `0.5`.

clone()
Deep copy of a `smmMutator`

2.4.4 Class `gsmMutator` (Function form: `GsmMutate`)

Generalized stepwise mutation model The *Generalized Stepwise Mutation model* (GSM) is an extension to the stepwise mutation model. This model assumes that alleles are represented by integer values and that a mutation either increases or decreases the allele value by a random value. In other words, in this model the change in the allelic state is drawn from a random distribution. A *geometric generalized stepwise model* uses a geometric distribution with parameter p, which has mean $\frac{p}{1-p}$ and variance $\frac{p}{(1-p)^2}$.

`gsmMutator` implements both models. If you specify a Python function without a parameter, this mutator will use its return value each time a mutation occur; otherwise, a parameter p should be provided and the mutator will act as a geometric generalized stepwise model.

class gsmMutator(*rate=[], loci=[], maxAllele=0, incProb=0.5, p=0, func=None, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a `gsmMutator`The GSM model is developed for allozymes. It provides better description for these kinds of evolutionary processes.

Please see class `mutator` for the descriptions of other parameters.

incProb: Probability to increase allele state. Default to `0.5`.

func: A function that returns the number of steps. This function does not accept any parameter.

clone()
Deep copy of a `gsmMutator`

mutate(*allele*)
Mutate according to the GSM model

2.4.5 Class `pyMutator` (Function form: `PyMutate`)

A hybrid mutator. Parameters such as mutation rate of this operator are set just like others and you are supposed to provide a Python function to return a new allele state given an old state. `pyMutator` will choose an allele as usual and call your function to mutate it to another allele.

class pyMutator (*rate=[], loci=[], maxAllele=0, func=None, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a `pyMutator`

clone()

Deep copy of a `pyMutator`

mutate(*allele*)

Mutate according to the mixed model

2.4.6 Class `pointMutator` (Function form: `PointMutate`)

Point mutator Mutate specified individuals at specified loci to a spcified allele. I.e., this is a non-random mutator used to introduce diseases etc. `pointMutator`, as its name suggest, does point mutation. This mutator will turn alleles at `loci` on the first chromosome copy to `toAllele` for individual `inds`. You can specify `atPloidy` to mutate other, or all ploidy copies.

class pointMutator (*loci, toAllele, atPloidy=[], inds=[], output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a `pointMutator` Please see class `mutator` for the descriptions of other parameters.

inds: Individuals who will mutate

toAllele: Allele that will be mutate to

apply(*pop*)

Apply a `pointMutator`

clone()

Deep copy of a `pointMutator`

mutationCount(*locus*)

Return mutation count at `locus`

mutationCounts()

Return mutation counts

2.5 Recombination and gene conversion

2.5.1 Class `recombinator`

In simuPOP, only one recombinator is provided. Recombination events between loci a/b and b/c are independent, otherwise there will be some linkage between loci. Users need to specify physical recombination rate between adjacent loci. In addition, for the recombinator

- it only works for diploid (and for females in haplodiploid) populations.
- the recombination rate must be comprised between `0.0` and `0.5`. A recombination rate of `0.0` means that the loci are completely linked, and thus behave together as a single linked locus. A recombination rate of `0.5` is equivalent to free of recombination. All other values between `0.0` and `0.5` will represent various linkage intensities between adjacent pairs of loci. The recombination rate is equivalent to `1-linkage` and represents the probability that the allele at the next locus is randomly drawn.
- it works for selfing. I.e., when only one parent is provided, it will be recombined twice, producing both maternal and paternal chromosomes of the offspring.
- conversion is allowed. Note that conversion will nullify many recombination events, depending on the parameters chosen.

class `recombinator` (*intensity=-1, rate=[], afterLoci=[], maleIntensity=-1, maleRate=[], maleAfterLoci=[], convProb=0, convMode=CONVERT_NumMarkers, convParam=1., begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Recombine chromosomes from parents

intensity: Intensity of recombination. The actual recombination rate between two loci is determined by `intensity*locus distance (between them)`.

rate: Recombination rate regardless of locus distance after all `afterLoci`. It can also be an array of recombination rates. Should have the same length as `afterLoci` or `totNumOfLoci()`. The recombination rates are independent of locus distance.

afterLoci: An array of locus indexes. Recombination will occur after these loci. If `rate` is also specified, they should have the same length. Default to all loci (but meaningless for those loci located at the end of a chromosome). If this parameter is given, it should be ordered, and can not include loci at the end of a chromosome.

maleIntensity: Recombination intensity for male individuals. If given, parameter `intensity` will be considered as female intensity.

maleRate: Recombination rate for male individuals. If given, parameter `rate` will be considered as female recombination rate.

maleAfterLoci: If given, males will recombine at different locations.

convProb: The probability of conversion event among all recombination events. When a recombination event happens, it may become a recombination event if the Holliday junction is resolved/repaired successfully, or a conversion event if the junction is not resolved/repaired. The default `convProb` is 0, meaning no conversion event at all. Note that the ratio of conversion to recombination events varies greatly from study to study, ranging from 0.1 to 15 (Chen et al, Nature Review Genetics, 2007). This translate to 0.1/0.9õ.1 to 15/16õ.94 of this parameter. When `convProb` is 1, all recombination events will be conversion events.

convMode: Conversion mode, determines how track length is determined.

- CONVERT_NumMarkers Converts a fixed number of markers.
- CONVERT_GeometricDistribution An geometric distribution is used to determine how many markers will be converted.
- CONVERT_TractLength Converts a fixed length of tract.
- CONVERT_ExponentialDistribution An exponential distribution with parameter `convLen` will be used to determine track length.

convParam: Parameter for the conversion process. The exact meaning of this parameter is determined by `convMode`. Note that

- conversion tract length is usually short, and is estimated to be between 337 and 456 bp, with overall range between maybe 50 - 2500 bp.
- simuPOP does not impose a unit for marker distance so your choice of `convParam` needs to be consistent with your unit. In the HapMap dataset, cM is usually assumed and marker distances are around 10kb (0.001cM ≈ 1kb). Gene conversion can largely be ignored. This is important when you use distance based conversion mode such as `CONVERT_TrackLength` or `CONVERT_ExponentialDistribution`.
- After a track length is determined, if a second recombination event happens within this region, the track length will be shortened. Note that conversion is identical to double recombination under this context.

haplodiploid: If set to true, the first copy of paternal chromosomes is copied directly as the paternal chromosomes of the offspring. This is because haplodiploid male has only one set of chromosome.

Note There is no recombination between sex chromosomes of male individuals if `sexChrom()=True`. This may change later if the exchanges of genes between pseudoautosomal regions of `XY` need to be modeled.

`clone`()

Deep copy of a recombinator

convCount (*size*)
Return the count of conversion of a certain size (only valid in standard modules)

convCounts ()
Return the count of conversions of all sizes (only valid in standard modules)

initialize (*pop*)
FIXME: No document

produceOffspring (*parent, off*)
Recombine parental chromosomes of *parent* and pass them to offspring *off*. The homologous chromosomes ofe parent will be recombined twice and form both homologous sets of the offspring, as if *parent* mates with itself (a selfing inheritance model). If sex chromosomes are present, offspring sex will be determined by which sex chromosomes are inherited by *off*. Random sex is assigned to *off* otherwise.

produceOffspring (*mom, dad, off*)
Recombine parental chromosomes and pass them to offspring *off*. A Mendelian inheritance model will be used, which recombine homologous sets of chromosomes of *mom* and *dad* and pass them as the first and second sets of homologous chromosomes to offspring *off*, respectively. If sex chromosomes are present, offspring sex is determined by which sex chromosomes are inherited by *off*. Random sex is assigned to *off* otherwise.

recCount (*locus*)
Return recombination count at a locus (only valid in standard modules)

recCounts ()
Return recombination counts (only valid in standard modules)

2.6 Selection

2.6.1 Class `selector`

A base selection operator for all selectors. Genetic selection is tricky to simulate since there are many different *fitness* values and many different ways to apply selection. simuPOP employs an *'ability-to-mate'* approach. Namely, the probability that an individual will be chosen for mating is proportional to its fitness value. More specifically,

- `PreMating` selectors assign fitness values to each individual, and mark part or all subpopulations as under selection.
- during sexless mating (e.g. `binomialSelection` mating scheme), individuals are chosen at probabilities that are proportional to their fitness values. If there are N individuals with fitness values $f_i, i = 1, ..., N$, individual i will have probability $\frac{f_i}{\sum_j f_j}$ to be chosen and passed to the next generation.
- during `randomMating`, males and females are separated. They are chosen from their respective groups in the same manner as `binomialSelection` and mate.

All of the selection operators, when applied, will set an information field `fitness` (configurable) and then mark part or all subpopulations as under selection. (You can use different selectors to simulate various selection intensities for different subpopulations). Then, a *'selector-aware'* mating scheme can select individuals according to their `fitness` information fields. This implies that

- only mating schemes can actually select individuals.
- a selector has to be a `PreMating` operator. This is not a problem when you use the operator form of the selector since its default stage is `PreMating`. However, if you use the function form of the selector in a `pyOperator`, make sure to set the stage of `pyOperator` to `PreMating`.

Note:

You can not apply two selectors to the same subpopulation, because only one fitness value is allowed for each individual.

class `selector`(*subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["fitness"]*)
Create a selector

subPop: A shortcut to `subPops=[subPop]`

subPops: Subpopulations that the selector will apply to. Default to all.

`apply`(*pop*)
Set fitness to all individuals. No selection will happen!

`clone`()
Deep copy of a selector

2.6.2 Class `mapSelector` (Function form: `MapSelector`, Applicable to `all ploidy`)

Selection according to the genotype at one or more loci This map selector implements selection according to genotype at one or more loci. A user provided dictionary (map) of genotypes will be used in this selector to set each individual's fitness value.

class `mapSelector`(*loci, fitness, phase=False, subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["fitness"]*)
Create a map selector

locus: The locus index. A shortcut to `loci=[locus]`

loci: The locus indexes. The genotypes at these loci will be used to determine the fitness value.

fitness: A dictionary of fitness values. The genotype must be in the form of `'a-b'` for a single locus, and `'a-b|c-d|e-f'` for multi-loci. In the haploid case, the genotype should be specified in the form of `'a'` for single locus, and `'a|b|c'` for multi-locus models.

phase: If `True`, genotypes `a-b` and `b-a` will have different fitness values. Default to `False`.

output: And other parameters please refer to help (`baseOperator.__init__`)

`clone`()
Deep copy of a map selector

`indFitness`(*ind, gen*)
Calculate/return the fitness value, currently assuming diploid

2.6.3 Class `maSelector` (Function form: `MaSelect`)

Multiple allele selector (selection according to wildtype or diseased alleles) This is called 'multiple-allele' selector. It separates alleles into two groups: wildtype and diseased alleles. Wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased alleles. This selector accepts an array of fitness values:

- For single-locus, `fitness` is the fitness for genotypes AA, Aa, aa, while A stands for wildtype alleles.
- For a two-locus model, `fitness` is the fitness for genotypes AABB, AABb, AAbb, AaBB, AbBb, Aabb, aaBB, aaBb and aaBb.
- For a model with more than two loci, use a table of length 3^n in a order similar to the two-locus model.

class maSelector (*loci, fitness, wildtype, subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["fitness"]*)

Create a multiple allele selector Please refer to `baseOperator` for other parameter descriptions.

fitness: For the single locus case, `fitness` is an array of fitness of AA, Aa, aa. A is the wildtype group. In the case of multiple loci, fitness should be in the order of AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, aabb.

wildtype: An array of alleles in the wildtype group. Any other alleles are considered to be diseased alleles. Default to `[0]`.

output: And other parameters please refer to help (`baseOperator.__init__`)

Note

- `maSelector` only works for diploid populations.
- `wildtype` alleles at all loci are the same.

clone()

Deep copy of a `maSelector`

indFitness(*ind, gen*)

Calculate/return the fitness value, currently assuming diploid

2.6.4 Class `mlSelector` (Function form: `MlSelect`)

Selection according to genotypes at multiple loci in a multiplicative model This selector is a 'multiple-locus model' selector. The selector takes a vector of selectors (can not be another `mlSelector`) and evaluate the fitness of an individual as the product or sum of individual fitness values. The mode is determined by parameter `mode`, which takes one of the following values

- `SEL_Multiplicative`: the fitness is calculated as $f = \prod_i f_i$, where f_i is the single-locus fitness value.
- `SEL_Additive`: the fitness is calculated as $f = \max(0, 1 - \sum_i (1 - f_i))$. f will be set to `0` when $f < 0$.

class mlSelector (*selectors, mode=SEL_Multiplicative, subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["fitness"]*)

Create a multiple-locus selector Please refer to `mapSelector` for other parameter descriptions.

selectors: A list of selectors

clone()

Deep copy of a `mlSelector`

indFitness(*ind, gen*)

Calculate/return the fitness value, currently assuming diploid

2.6.5 Class `pySelector` (Function form: `PySelect`)

Selection using user provided function This selector assigns fitness values by calling a user provided function. It accepts a list of loci and a Python function `func`. For each individual, this operator will pass the genotypes at these loci, generation number, and optionally values at some information fields to this function. The return value is treated as the fitness value. The genotypes are arranged in the order of `0-0,0-1,1-0,1-1` etc. where X-Y represents locus X - ploidy Y. More specifically, `func` can be

- `func(geno, gen)` if `infoFields` has length 0 or 1.
- `func(geno, gen, fields)` when `infoFields` has more than 1 fields. Values of fields 1, 2, ... will be passed. Both `geno` and `fields` should be a list.

class pySelector(*loci, func, subPops=[], stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["fitness"]*)

Create a Python hybrid selector

loci: Susceptibility loci. The genotype at these loci will be passed to `func`.

func: A Python function that accepts genotypes at specified loci, generation number, and optionally information fields. It returns the fitness value.

output: And other parameters please refer to help (`baseOperator.__init__`)

infoFields: If specified, the first field should be the information field to save calculated fitness value (should be 'fitness' in most cases). The values of the rest of the information fields (if available) will also be passed to the user defined penetrance function.

clone()

Deep copy of a `pySelector`

indFitness(*ind, gen*)

Calculate/return the fitness value, currently assuming diploid

2.7 Penetrance

2.7.1 Class `penetrance`

Base class of all penetrance operators. Penetrance is the probability that one will have the disease when he has certain genotype(s). An individual will be randomly marked as affected/unaffected according to his/her penetrance value. For example, an individual will have probability 0.8 to be affected if the penetrance is 0.8.

Penetrance can be applied at any stage (default to `DuringMating`). When a penetrance operator is applied, it calculates the penetrance value of each offspring and assigns affected status accordingly. Penetrance can also be used `PreMating` or `PostMating`. In these cases, the affected status will be set to all individuals according to their penetrance values.

Penetrance values are usually not saved. If you would like to know the penetrance value, you need to

- use `addInfoField('penetrance')` to the population to analyze. (Or use `infoFields` parameter of the population constructor), and
- use e.g., `mlPenetrance(...., infoFields=['penetrance'])` to add the penetrance field to the penetrance operator you use. You may choose a name other than `'penetrance'` as long as the field names for the operator and population match.

Penetrance functions can be applied to the current, all, or certain number of ancestral generations. This is controlled by the `ancestralGen` parameter, which is default to `-1` (all available ancestral generations). You can set it to `0` if you only need affection status for the current generation, or specify a number `n` for the number of ancestral generations (n + 1 total generations) to process. Note that the `ancestralGen` parameter is ignored if the penetrance operator is used as a during mating operator.

class penetrance(*ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a penetrance operator

ancestralGen: If this parameter is set to be `0`, apply penetrance to the current generation; if `-1`, apply to all generations; otherwise, apply to the specified numbers of ancestral generations.

stage: Specify the stage this operator will be applied. Default to `DuringMating`.

infoFields: If one field is specified, it will be used to store penetrance values.

apply(*pop*)
Set penetrance to all individuals and record penetrance if requested

clone()
Deep copy of a penetrance operator

penet()
Calculate/return penetrance etc.

2.7.2 Class `mapPenetrance` (Function form: `MapPenetrance`)

Penetrance according to the genotype at one locus Assign penetrance using a table with keys 'X-Y' where X and Y are allele numbers.

class mapPenetrance(*loci, penet, phase=False, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a map penetrance operator

locus: The locus index. Shortcut to `loci=[locus]`

loci: The locus indexes. The genotypes of these loci will be used to determine penetrance.

penet: A dictionary of penetrance. The genotype must be in the form of 'a-b' for a single locus.

phase: If `True`, `a/b` and `b/a` will have different penetrance values. Default to `False`.

output: And other parameters please refer to help(baseOperator.__init__)

clone()
Deep copy of a map penetrance operator

2.7.3 Class `maPenetrance` (Function form: `MaPenetrance`)

Multiple allele penetrance operator This is called 'multiple-allele' penetrance. It separates alleles into two groups: wildtype and diseased alleles. Wildtype alleles are specified by parameter `wildtype` and any other alleles are considered as diseased alleles. `maPenetrance` accepts an array of penetrance for AA, Aa, aa in the single-locus case, and a longer table for the multi-locus case. Penetrance is then set for any given genotype.

class maPenetrance(*loci, penet, wildtype, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a multiple allele penetrance operator (penetrance according to diseased or wildtype alleles)

locus: The locus index. The genotype of this locus will be used to determine penetrance.

loci: The locus indexes. The genotypes of these loci will be examed.

penet: An array of penetrance values of AA, Aa, aa. A is the wild type group. In the case of multiple loci, penetrance should be in the order of AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, aabb.

wildtype: An array of alleles in the wildtype group. Any other alleles will be considered as in the diseased allele group.

output: And other parameters please refer to help(baseOperator.__init__)

clone()
Deep copy of a multi-allele penetrance operator

penet(*ind*)
Currently assuming diploid

2.7.4 Class `mlPenetrance` (Function form: `MlPenetrance`)

Penetrance according to the genotype according to a multiple loci multiplicative model This is the 'multiple-locus' penetrnace calculator. It accepts a list of penetrances and combine them according to the `mode` parameter, which takes one of the following values:

- `PEN_Multiplicative`: the penetrance is calculated as $f = \prod f_i$.
- `PEN_Additive`: the penetrance is calculated as $f = \min(1, \sum f_i)$. f will be set to `1` when $f < 0$. In this case, s_i are added, not f_i directly.
- `PEN_Heterogeneity`: the penetrance is calculated as $f = 1 - \prod (1 - f_i)$.

Please refer to Neil Risch (1990) for detailed information about these models.

class `mlPenetrance`(*peneOps, mode=PEN_Multiplicative, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a multiple locus penetrance operator

peneOps: A list of penetrance operators

mode: Can be one of `PEN_Multiplicative`, `PEN_Additive`, and `PEN_Heterogeneity`

`clone`()

Deep copy of a multi-loci penetrance operator

`penet`(*ind*)

Currently assuming diploid

2.7.5 Class `pyPenetrance` (Function form: `PyPenetrance`)

Assign penetrance values by calling a user provided function For each individual, the penetrance is determined by a user-defined penetrance function `func`. This function takes genetypes at specified loci, and optionally values of specified information fields. The return value is considered as the penetrance for this individual. More specifically, `func` can be

- `func(geno)` if `infoFields` has length 0 or 1.
- `func(geno, fields)` when `infoFields` has more than 1 fields. Both parameters should be an list.

class `pyPenetrance`(*loci, func, ancestralGen=-1, stage=DuringMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Provide locus and penetrance for 11, 12, 13 (in the form of dictionary)

loci: The genotypes at these loci will be passed to the provided Python function in the form of `loc1_1, loc1_2, loc2_1, loc2_2, ...` if the individuals are diploid.

func: A user-defined Python function that accepts an array of genotypes at specified loci and return a penetrance value. The return value should be between `0` and `1`.

infoFields: If specified, the first field should be the information field to save calculated penetrance value. The values of the rest of the information fields (if available) will also be passed to the user defined penetrance function.

output: And other parameters please refer to help (`baseOperator.__init__`)

`clone`()

Deep copy of a Python penetrance operator

`penet`(*ind*)

Currently assuming diploid

2.8 Quantitative Trait

2.8.1 Class `quanTrait`

Base class of quantitative trait Quantitative trait is the measure of certain phenotype for given genotype. Quantitative trait is similar to penetrance in that the consequence of penetrance is binary: affected or unaffected; while it is continuous for quantitative trait.

In simuPOP, different operators or functions were implemented to calculate quantitative traits for each individual and store the values in the information fields specified by the user (default to `qtrait`). The quantitative trait operators also accept the `ancestralGen` parameter to control the number of generations for which the `qtrait` information field will be set.

class `quanTrait` (*ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["qtrait"]*)

Create a quantitative trait operator

`apply` (*pop*)

Set `qtrait` to all individual

`clone` ()

Deep copy of a quantitative trait operator

`qtrait` ()

Calculate/return quantitative trait etc.

2.8.2 Class `mapQuanTrait` (Function form: `MapQuanTrait`)

Quantitative trait according to genotype at one locus Assign quantitative trait using a table with keys 'X-Y' where X and Y are allele numbers. If parameter `sigma` is not zero, the return value is the sum of the trait plus $N\left(0,\sigma^2\right)$. This random part is usually considered as the environmental factor of the trait.

class `mapQuanTrait` (*loci, qtrait, sigma=0, phase=False, ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["qtrait"]*)

Create a map quantitative trait operator

locus: The locus index. The quantitative trait is determined by genotype at this locus.

loci: An array of locus indexes. The quantitative trait is determined by genotypes at these loci.

qtrait: A dictionary of quantitative traits. The genotype must be in the form of 'a-b'. This is the mean of the quantitative trait. The actual trait value will be $N\left(mean,\sigma^2\right)$. For multiple loci, the form is 'a-blc-dle-f' etc.

sigma: Standard deviation of the environmental factor $N\left(0,\sigma^2\right)$.

phase: If `True`, a/b and b/a will have different quantitative trait values. Default to `False`.

output: And other parameters please refer to help (`baseOperator.__init__`)

`clone` ()

Deep copy of a map quantitative trait operator

`qtrait` (*ind*)

Currently assuming diploid

2.8.3 Class `maQuanTrait` (Function form: `MaQuanTrait`)

Multiple allele quantitative trait (quantitative trait according to disease or wildtype alleles) This is called 'multiple-allele' quantitative trait. It separates alleles into two groups: wildtype and diseased alleles. Wildtype alleles are

specified by parameter `wildtype` and any other alleles are considered as diseased alleles. `maQuanTrait` accepts an array of fitness. Quantitative trait is then set for any given genotype. A standard normal distribution $N\left(0,\sigma^2\right)$ will be added to the returned trait value.

class maQuanTrait(*loci, qtrait, wildtype, sigma=[], ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["qtrait"]*)

Create a multiple allele quantitative trait operator Please refer to `quanTrait` for other parameter descriptions.

qtrait: An array of quantitative traits of AA, Aa, aa. A is the wildtype group

sigma: An array of standard deviations for each of the trait genotype (AA, Aa, aa)

wildtype: An array of alleles in the wildtype group. Any other alleles will be considered as diseased alleles. Default to `[0]`.

output: And other parameters please refer to help(`baseOperator.__init__`)

clone()

Deep copy of a multiple allele quantitative trait

qtrait(*ind*)

Currently assuming diploid

2.8.4 Class `mlQuanTrait` (Function form: `MlQuanTrait`)

Quantitative trait according to genotypes from a multiple loci multiplicative model Operator `mlQuanTrait` is a 'multiple-locus' quantitative trait calculator. It accepts a list of quantitative traits and combine them according to the `mode` parameter, which takes one of the following values

- `QT_Multiplicative`: the mean of the quantitative trait is calculated as $f=\prod f_i$.
- `QT_Additive`: the mean of the quantitative trait is calculated as $f=\sum f_i$.

Note that all σ_i (for f_i) and σ (for f) will be considered. I.e, the trait value should be

$$f=\sum_i\left(f_i+N\left(0,\sigma_i^2\right)\right)+\sigma^2$$

for `QT_Additive` case. If this is not desired, you can set some of the σ to zero.

class mlQuanTrait(*qtraits, mode=QT_Multiplicative, sigma=0, ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["qtrait"]*)

Create a multiple locus quantitative trait operator Please refer to `quanTrait` for other parameter descriptions.

qtraits: A list of quantitative traits

mode: Can be one of `QT_Multiplicative` and `QT_Additive`

clone()

Deep copy of a multiple loci quantitative trait operator

qtrait(*ind*)

Currently assuming diploid

2.8.5 Class `pyQuanTrait` (Function form: `PyQuanTrait`)

Quantitative trait using a user provided function For each individual, a user provided function is used to calculate quantitative trait.

class pyQuanTrait (*loci, func, ancestralGen=-1, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=["qtrait"]*)

Create a Python quantitative trait operator Please refer to `quanTrait` for other parameter descriptions.

loci: The genotypes at these loci will be passed to `func`.

func: A Python function that accepts genotypes at specified loci and returns the quantitative trait value.

output: And other parameters please refer to help(`baseOperator.__init__`)

clone()

Deep copy of a Python quantitative trait operator

qtrait(*ind*)

Currently assuming diploid

2.9 Ascertainment

2.10 Statistics Calculation

2.10.1 Class `stator`

Base class of all the statistics calculator Operator `stator` calculates various basic statistics for the population and set variables in the local namespace. Other operators or functions can refer to the results from the namespace after `stat` is applied.

class stator (*output="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a stator

clone()

Deep copy of a stator

2.10.2 Class `stat` (Function form: `Stat`)

Calculate statistics Operator `stat` calculates various basic statistics for the population and sets variables in the local namespace. Other operators or functions can refer to the results from the namespace after `stat` is applied. `Stat` is the function form of the operator.

Note that these statistics are dependent to each other. For example, heterotype and allele frequencies of related loci will be automatically calculated if linkage diseqilibrium is requested.

class stat (*popSize=False, numOfMale=False, numOfMale_param={}, numOfAffected=False, numOfAffected_param={}, numOfAlleles=[], numOfAlleles_param={}, alleleFreq=[], alleleFreq_param={}, heteroFreq=[], expHetero=[], expHetero_param={}, homoFreq=[], genoFreq=[], genoFreq_param={}, haploFreq=[], LD=[], LD_param={}, association=[], association_param={}, Fst=[], Fst_param={}, relGroups=[], relLoci=[], rel_param={}, relBySubPop=False, relMethod=[], relMinScored=10, hasPhase=False, midValues=False, output="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create an `stat` operator

popSize: Whether or not calculate population and virtual subpopulation sizes. This parameter will set the following variables:

- `numSubPop` the number of subpopulations.
- `subPopSize` an array of subpopulation sizes.

- `virtualSubPopSize` (optional) an array of virtual subpopulation sizes. If a subpopulation does not have any virtual subpopulation, the subpopulation size is returned.
- `popSize`, `subPop[sp]['popSize']` the population/subpopulation size.

numOfMale: Whether or not count the numbers or proportions of males and females. This parameter can set the following variables by user's specification:

- `numOfMale`, `subPop[sp]['numOfMale']` the number of males in the population/subpopulation.
- `numOfFemale`, `subPop[sp]['numOfFemale']` the number of females in the population/subpopulation.
- `propOfMale`, `subPop[sp]['propOfMale']` the proportion of males in the population/subpopulation.
- `propOfFemale`, `subPop[sp]['propOfFemale']` the proportion of females in the population/subpopulation.

numOfMale_param: A dictionary of parameters of `numOfMale` statistics. Can be one or more items choosen from the following options: `numOfMale`, `propOfMale`, `numOfFemale`, and `propOfFemale`.

numOfAffected: Whether or not count the numbers or proportions of affected and unaffected individuals. This parameter can set the following variables by user's specification:

- `numOfAffected`, `subPop[sp]['numOfAffected']` the number of affected individuals in the population/subpopulation.
- `numOfUnaffected`, `subPop[sp]['numOfUnAffected']` the number of unaffected individuals in the population/subpopulation.
- `propOfAffected`, `subPop[sp]['propOfAffected']` the proportion of affected individuals in the population/subpopulation.
- `propOfUnaffected`, `subPop[sp]['propOfUnAffected']` the proportion of unaffected individuals in the population/subpopulation.

numOfAffected_param: A dictionary of parameters of `numOfAffected` statistics. Can be one or more items choosen from the following options: `numOfAffected`, `propOfAffected`, `numOfUnaffected`, `propOfUnaffected`.

numOfAlleles: An array of loci at which the numbers of distinct alleles will be counted (`numOfAlleles=[loc1, loc2, ...]` where `loc1` etc. are absolute locus indexes). This is done through the calculation of allele frequencies. Therefore, allele frequencies will also be calculated if this statistics is requested. This parameter will set the following variables (`carray` objects of the numbers of alleles for *all loci*). Unrequested loci will have `0` distinct alleles.

- `numOfAlleles`, `subPop[sp]['numOfAlleles']` the number of distinct alleles at each locus. (Calculated only at requested loci.)

numOfAlleles_param: A dictionary of parameters of `numOfAlleles` statistics. Can be one or more items choosen from the following options: `numOfAffected`, `propOfAffected`, `numOfUnaffected`, `propOfUnaffected`.

alleleFreq: An array of loci at which all allele frequencies will be calculated (`alleleFreq=[loc1, loc2, ...]` where `loc1` etc. are loci where allele frequencies will be calculated). This parameter will set the following variables (`carray` objects); for example, `alleleNum[1][2]` will be the number of allele `2` at locus `1`:

- `alleleNum[a]`, `subPop[sp]['alleleNum'][a]`
- `alleleFreq[a]`, `subPop[sp]['alleleFreq'][a]`.

alleleFreq_param: A dictionary of parameters of `alleleFreq` statistics. Can be one or more items choosen from the following options: `numOfAlleles`, `alleleNum`, and `alleleFreq`.

genoFreq: An array of loci at which all genotype frequencies will be calculated (`genoFreq=[loc1, loc2, ...]`. You may use parameter `genoFreq_param` to control if `a/b` and `b/a` are the same genotype. This parameter will set the following dictionary variables. Note that unlike list used for `alleleFreq` etc., the indexes `a`, `b` of `genoFreq[loc][a][b]` are dictionary keys, so you will get a *KeyError* when you used a wrong key. You can get around this problem by using expressions like `genoNum[loc].setDefault(a,{})`.

- `genoNum[loc][allele1][allele2]` and `subPop[sp]['genoNum'][loc][allele1][allele2]`, the number of genotype `allele1-allele2` at locus `loc`.
- `genoFreq[loc][allele1][allele2]` and `subPop[sp]['genoFreq'][loc][allele1][allele2]`, the frequency of genotype `allele1-allele2` at locus `loc`.
- genoFreq_param a dictionary of parameters of `phase` = 0 or 1.

heteroFreq: An array of loci at which observed heterozygosities will be calculated (`heteroFreq=[loc1, loc2, ...]`). For each locus, the number and frequency of allele specific and overall heterozygotes will be calcuated and stored in four population variables. For example, `heteroNum[loc][1]` stores number of heterozygotes at locus `loc`, with respect to allele `1`, which is the number of all genotype `1x` or `x1` where does not equal to `1`. All other genotypes such as `02` are considered as homozygotes when `heteroFreq[loc][1]` is calculated. The overall number of heterozygotes (`HeteroNum[loc]`) is the number of genotype `xy` if `x` does not equal to `y`.

- `HeteroNum[loc]`, `subPop[sp]['HeteroNum'][loc]`, the overall heterozygote count.
- `HeteroFreq[loc]`, `subPop[sp]['HeteroFreq'][loc]`, the overall heterozygote frequency.
- `heteroNum[loc][allele]`, `subPop[sp]['heteroNum'][loc][allele]`, allele-specific heterozygote counts.
- `heteroFreq[loc][allele]`, `subPop[sp]['heteroFreq'][loc][allele]`, allele-specific heterozygote frequency.

homoFreq: An array of loci to calculate observed homozygosities and expected homozygosities (`homoFreq=[loc1, loc2, ...]`). This parameter will calculate the numbers and frequencies of homozygotes **xx** and set the following variables:

- `homoNum[loc]`, `subPop[sp]['homoNum'][loc]`.
- `homoFreq[loc]`, `subPop[sp]['homoFreq'][loc]`.

expHetero: An array of loci at which the expected heterozygosities will be calculated (`expHetero=[loc1, loc2, ...]`). The expected heterozygosity is calculated by

$$h_{exp} = 1 - p_i^2,$$

where p_i is the allele frequency of allele i. The following variables will be set:

- `expHetero[loc]`, `subPop[sp]['expHetero'][loc]`.

expHetero_param: A dictionary of parameters of `expHetero` statistics. Can be one or more items choosen from the following options: `subpop` and `midValues`.

haploFreq: A matrix of haplotypes (allele sequences on different loci) to count. For example, `haploFreq = [ [ 0,1,2 ], [1,2] ]` will count all haplotypes on loci 0, 1 and 2; and all haplotypes on loci 1, 2. If only one haplotype is specified, the outer `[]` can be omitted. I.e., `haploFreq=[0,1]` is acceptable. The following dictionary variables will be set with keys `0-1-2` etc. For example, `haploNum['1-2']['5-6']` is the number of allele pair 5, 6 (on loci 1 and 2 respectively) in the population.

- `haploNum[haplo]` and `subPop[sp]['haploNum'][haplo]`, the number of allele sequencies on loci `haplo`.
- `haploFreq[haplo]`, `subPop[sp]['haploFreq'][haplo]`, the frequency of allele sequencies on loci `haplo`.

LD: Calculate linkage disequilibria LD, LD' and r^2, given `LD=[ [loc1, loc2], [ loc1, loc2, allele1, allele2], ... ]`. For each item `[loc1, loc2, allele1, allele2]`, D, D' and r^2 will be calculated based on `allele1` at `loc1` and `allele2` at `loc2`. If only two loci are given, the LD values are averaged over all allele pairs. For example, for allele A at locus `1` and allele B at locus `2`,

$$D = P_{AB} - P_A P_B$$

$$D' = D/D_{max}$$

$$D_{max} = \min\left(P_A\left(1-P_B\right), \left(1-P_A\right)P_B\right) \text{ if } D > 0 \; \min\left(P_A P_B, \left(1-P_A\right)\left(1-P_B\right)\right) \text{ if } D < 0$$

$$r^2 = \frac{D^2}{P_A\left(1-P_A\right)P_B\left(1-P_B\right)}$$

If only one item is specified, the outer `[]` can be ignored. I.e., `LD=[loc1, loc2]` is acceptable. This parameter will set the following variables. Please note that the difference between the data structures used for `ld` and `LD`.

- `ld['loc1-loc2']['allele1-allele2']`, `subPop[sp]['ld']['loc1-loc2']['allele1-allele2']`
- `ld_prime['loc1-loc2']['allele1-allele2']`, `subPop[sp]['ld_prime']['loc1-loc2']['allel`
- `r2['loc1-loc2']['allele1-allele2']`, `subPop[sp]['r2']['loc1-loc2']['allele1-allele2']`
- `LD[loc1][loc2]`, `subPop[sp]['LD'][loc1][loc2]`.
- `LD_prime[loc1][loc2]`, `subPop[sp]['LD_prime'][loc1][loc2]`.
- `R2[loc1][loc2]`, `subPop[sp]['R2'][loc1][loc2]`.

LD_param: A dictionary of parameters of `LD` statistics. Can have key `stat` which is a list of statistics to calculate. Default to all. If any statistics is specified, only those specified will be calculated. For example, you may use `LD_param={LD_prime}` to calculate D' only, where `LD_prime` is a shortcut for `'stat':['LD_prime']`. Other parameters that you may use are:

- `subPop` whether or not calculate statistics for subpopulations.
- `midValues` whether or not keep intermediate results.

association: Association measures

association_param: A dictionary of parameters of `association` statistics. Can be one or more items choosen from the following options: `ChiSq`, `ChiSq_P`, `UC_U`, and `CramerV`.

Fst: Calculate F_{st}, F_{is}, F_{it}. For example, `Fst = [0,1,2]` will calculate F_{st}, F_{is}, F_{it} based on alleles at loci `0`, `1`, `2`. The locus-specific values will be used to calculate `AvgFst`, which is an average value over all alleles (Weir & Cockerham, 1984). Terms and values that match Weir & Cockerham are:

- F (F_{IT}) the correlation of genes within individuals (inbreeding);
- θ (F_{ST}) the correlation of genes of difference individuals in the same population (will evaluate for each subpopulation and the whole population)
- f (F_{IS}) the correlation of genes within individuals within populations.

This parameter will set the following variables:

- `Fst[loc]`, `Fis[loc]`, `Fit[loc]`
- `AvgFst`, `AvgFis`, `AvgFit`.

Fst_param: A dictionary of parameters of `Fst` statistics. Can be one or more items choosen from the following options: `Fst`, `Fis`, `Fit`, `AvgFst`, `AvgFis`, and `AvgFit`.

relMethod: Method used to calculate relatedness. Can be either `REL_Queller` or `REL_Lynch`. The relatedness values between two individuals, or two groups of individuals are calculated according to Queller & Goodnight (1989) (`method=REL_Queller`) and Lynch et al. (1999) (`method=REL_Lynch`). The results are pairwise relatedness values, in the form of a matrix. Original group or subpopulation numbers are discarded. There is no subpopulation level relatedness value.

relGroups: Calculate pairwise relatedness between groups. Can be in the form of either `[[1,2,3],[5,6,7],[8,9]]` or `[2,3,4]`. The first one specifies groups of individuals, while the second specifies subpopulations. By default, relatedness between subpopulations is calculated.

relLoci: Loci on which relatedness values are calculated

rel_param: A dictionary of parameters of relatedness statistics. Can be one or more items choosen from the following options: `Fst`, `Fis`, `Fit`, `AvgFst`, `AvgFis`, and `AvgFit`.

hasPhase: If a/b and b/a are the same genotype. Default to `False`.

midValues: Whether or not post intermediate results. Default to `False`. For example, `Fst` will need to calculate allele frequencise. If `midValues` is set to `True`, allele frequencies will be posted as well. This will be helpful in debugging and sometimes in deriving statistics.

apply(*pop*)
Apply the `stat` operator

clone()
Deep copy of a `stat` operator

2.11 Expression and Statements

2.11.1 Class `dumper`

Dump the content of a population.

class dumper(*alleleOnly=False, infoOnly=False, ancestralPops=False, dispWidth=1, max=100, chrom=[], loci=[], subPop=[], indRange=[], output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Dump a population

alleleOnly: Only display allele

infoOnly: Only display genotypic information

dispWidth: Number of characters to display an allele. Default to `1`.

ancestralPops: Whether or not display ancestral populations. Default to `False`.

chrom: Chromosome(s) to display

loci: Loci to display

subPop: Only display subpopulation(s)

indRange: Range(s) of individuals to display

max: The maximum number of individuals to display. Default to `100`. This is to avoid careless dump of huge populations.

output: Output file. Default to the standard output.

outputExpr: And other parameters: refer to help(baseOperator.__init__)

alleleOnly()
Only show alleles (not structure, gene information?

apply(*pop*)
Apply to one population. It does not check if the operator is activated.

clone()
Deep copy of an operator

infoOnly()
Only show info

setAlleleOnly(*alleleOnly*)
FIXME: No document

setInfoOnly(*infoOnly*)
FIXME: No document

2.11.2 Class `savePopulation`

Save population to a file

class savePopulation(*output="", outputExpr="", format="", compress=True, stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Save population

output: Output filename.

outputExpr: An expression that will be evalulated dynamically to determine file name. Parameter `output` will be ignored if this parameter is given.

format: Obsolete parameter

compress: Obsolete parameter

apply(*pop*)
Apply to one population. It does not check if the operator is activated.

clone()
Deep copy of an operator

2.11.3 Class `pyOutput`

Output a given string. A common usage is to output a new line for the last replicate.

class pyOutput(*str="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a `pyOutput` operator that outputs a given string.

str: String to be outputted

apply(*pop*)
Simply output some info

clone()
Deep copy of an operator

setString(*str*)
Set output string.

2.11.4 Class `pyEval` (Function form: `PyEval`)

Evaluate an expression Python expressions/statements will be executed when `pyEval` is applied to a population by using parameters `expr/stmts`. Statements can also been executed when `pyEval` is created and destroyed or before

expr is executed. The corresponding parameters are preStmts, postStmts and stmts. For example, operator varPlotter uses this feature to initialize R plots and save plots to a file when finished.

class pyEval(*expr="", stmts="", preStmts="", postStmts="", exposePop=False, name="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Evaluate expressions/statments in the local namespace of a replicate

expr: The expression to be evaluated. The result will be sent to output.

stmts: The statement that will be executed before the expression

preStmts: The statement that will be executed when the operator is constructed

postStmts: The statement that will be executed when the operator is destroyed

exposePop: If True, expose the current population as a variable named pop

name: Used to let pure Python operator to identify themselves

output: Default to >. I.e., output to standard output.

apply(*pop*)

Apply the pyEval operator

clone()

Deep copy of a pyEval operator

name()

Return the name of an expression The name of a pyEval operator is given by an optional parameter name. It can be used to identify this pyEval operator in debug output, or in the dryrun mode of simulator::evolve.

2.11.5 Class pyExec (Function form: PyExec)

Execute a Python statement This operator takes a list of statements and executes them. No value will be returned or outputted.

class pyExec(*stmts="", preStmts="", postStmts="", exposePop=False, name="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Evaluate statments in the local replicate namespace, no return value Please refer to class pyEval for parameter descriptions.

clone()

Deep copy of a pyExec operator

2.11.6 Class infoEval (Function form: infoEval)

Unlike operator pyEval and pyExec that work at the population level, in its local namespace, infoEval works at the individual level, working with individual information fields. is statement can change the value of existing information fields. Optionally, variables in population's local namespace can be used in the statement, but this should be used with caution.

class infoEval(*expr="", stmts="", subPops=[], usePopVars=False, exposePop=False, name="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Evaluate Python statements with variables being an individual's information fields The expression and statements will be executed for each individual, in a Python namespace (dictionary) where individual information fields are made available as variables. Population dictionary can be made avaialbe with option usePopVars. Changes to these variables will change the corresponding information fields of individuals. Please note that, 1. If population variables are used, and there are name conflicts between information fields and variables, population variables will be overridden by information fields, without any warning. 2. Information fields are float

numbers. An exceptions will raise if an information field can not be converted to a float number. 3. This operator can be used in all stages. When it is used during-mating, it will act on each offspring.

expr: The expression to be evaluated. The result will be sent to `output`.

stmts: The statement that will be executed before the expression

subPop: A shortcut to `subPops=[subPop]`

subPops: Subpopulations this operator will apply to. Default to all.

usePopVars: If `True`, import variables from expose the current population as a variable named `pop`

exposePop: If `True`, expose the current population as a variable named `pop`

name: Used to let pure Python operator to identify themselves

output: Default to >. I.e., output to standard output. Note that because the expression will be executed for each individual, the output can be large.

apply(*pop*)
Apply the `infoEval` operator

clone()
Deep copy of a `infoEval` operator

name()
Return the name of an expression The name of a `infoEval` operator is given by an optional parameter `name`. It can be used to identify this `infoEval` operator in debug output, or in the dryrun mode of simulator::evolve.

2.11.7 Class `infoExec` (Function form: `infoExec`)

Execute a Python statement for each individual, using information fields This operator takes a list of statements and executes them. No value will be returned or outputted.

class infoExec(*stmts="", subPops=[], usePopVars=False, exposePop=False, name="", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Fields, optionally with variable in population's local namespace Please refer to class `infoEval` for parameter descriptions.

clone()
Deep copy of a `infoExec` operator

2.12 Tagging (used for pedigree tracking)

2.12.1 Class `tagger`

Base class of tagging individuals This is a during-mating operator that tags individuals with various information. Potential usages are:

- recording the parental information to track pedigree;
- tagging an individual/allele and monitoring its spread in the population etc.

class tagger(*output="", outputExpr="", begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a `tagger`, default to be always active but no output

apply(*pop*)
Add a newline

clone()
Deep copy of a
tagger

2.12.2 Class `inheritTagger`

Inherite tag from parents This during-mating operator will copy the tag (information field) from his/her parents. Depending on `mode` parameter, this tagger will obtain tag, value of the first specified information fields, from his/her father or mother (two tag fields), or both (first tag field from father, and second tag field from mother).

An example may be tagging one or a few parents and examining, at the last generation, how many offspring they have.

class inheritTagger(*mode=TAG_Paternal, begin=0, end=-1, step=1, at=[], rep=[], output="", outputExpr="", infoFields=["paternal_tag", "maternal_tag"]*)
Create an `inheritTagger` that inherits a tag from one or both parents

mode: Can be one of `TAG_Paternal`, `TAG_Maternal`, and `TAG_Both`

applyDuringMating(*pop, offspring, dad=None, mom=None*)
Apply the `inheritTagger`

clone()
Deep copy of a `inheritTagger`

2.12.3 Class `parentTagger`

Tagging according to parental indexes This during-mating operator set `tag()` each individual with indexes of his/her parent in the parental population. Because only one parent is recorded, this is recommended to be used for mating schemes that requires only one parent (such as selfMating). This tagger record indexes to information field parent_idx, and/or a given file. The usage is similar to parentsTagger.

class parentTagger(*begin=0, end=-1, step=1, at=[], rep=[], output="", outputExpr="", infoFields=["parent_idx"]*)
Create a `parentTagger`

apply(*pop*)
With a newline.

applyDuringMating(*pop, offspring, dad=None, mom=None*)
Apply the `parentTagger`

clone()
Deep copy of a `parentTagger`

2.12.4 Class `parentsTagger`

Tagging according to parents' indexes This during-mating operator set `tag()`, currently a pair of numbers, of each individual with indexes of his/her parents in the parental population. This information will be used by pedigree-related operators like `affectedSibpairSample` to track the pedigree information. Because parental population will be discarded or stored after mating, these index will not be affected by post-mating operators. This tagger record parental index to one or both

- one or two information fields. Default to father_idx and mother_idx. If only one parent is passed in a mating scheme (such as selfing), only the first information field is used. If two parents are passed, the first information field records paternal index, and the second records maternal index.

- a file. Indexes will be written to this file. This tagger will also act as a post-mating operator to add a new-line to this file.

class parentsTagger(*begin=0, end=-1, step=1, at=[], rep=[], output="", outputExpr="", infoFields=["father_idx", "mother_idx"]*)
Create a `parentsTagger`

apply(*pop*)
With a newline.

applyDuringMating(*pop, offspring, dad=None, mom=None*)
Apply the `parentsTagger`

clone()
Deep copy of a `parentsTagger`

2.12.5 Class `sexTagger`

Tagging sex status. This is a simple post-mating tagger that write sex status to a file. By default, 1 for Male, 2 for Female.

class sexTagger(*code=[], begin=0, end=-1, step=1, at=[], rep=[], stage=PostMating, output=">", outputExpr="", infoFields=[]*)
FIXME: No document

code: Code for Male and Female, default to 1 and 2, respectively. This is used by Linkage format.

apply(*pop*)
Add a newline

2.12.6 Class `affectionTagger`

Tagging affection status. This is a simple post-mating tagger that write affection status to a file. By default, 1 for unaffected, 2 for affected.

class affectionTagger(*code=[], begin=0, end=-1, step=1, at=[], rep=[], stage=PostMating, output=">", outputExpr="", infoFields=[]*)
FIXME: No document

code: Code for Male and Female, default to 1 and 2, respectively. This is used by Linkage format.

apply(*pop*)
Add a newline

2.12.7 Class `infoTagger`

Tagging information fields. This is a simple post-mating tagger that write given information fields to a file (or standard output).

class infoTagger(*begin=0, end=-1, step=1, at=[], rep=[], stage=PostMating, output=">", outputExpr="", infoFields=[]*)
FIXME: No document

apply(*pop*)
Add a newline

2.12.8 Class `pyTagger`

Python tagger. This tagger takes some information fields from both parents, pass to a Python function and set the individual field with the return value. This operator can be used to trace the inheritance of trait values.

class `pyTagger` (*func=None, begin=0, end=-1, step=1, at=[], rep=[], output="", outputExpr="", infoFields=[]*)
Creates a `pyTagger` that works on specified information fields

infoFields: Information fields. The user should gurantee the existence of these fields.

func: A Pyton function that returns a list to assign the information fields. e.g., if `fields=['A', 'B']`, the function will pass values of fields `'A'` and `'B'` of father, followed by mother if there is one, to this function. The return value is assigned to fields `'A'` and `'B'` of the offspring. The return value has to be a list even if only one field is given.

`applyDuringMating` (*pop, offspring, dad=None, mom=None*)
Apply the `pyTagger`

`clone` ()
Deep copy of a `pyTagger`

2.13 Terminator

2.13.1 Class `terminateIf`

This operator evaluates an expression in a population's local namespace and terminate the evolution of this population, or the whole simulator, if the return value of this expression is `True`. Termination caused by an operator will stop the execution of all operators after it. Because a life-cycle is considered to be complete if mating is complete, the *evolved generations* (return value from `simulator::evolve`) of a terminated replicate is determined by when the last evolution cycle is terminated.

class `terminateIf` (*condition="", stopAll=False, message="", output="", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a terminator with an expression *condition*, which will be evalulated in a population's local namespace when the operator is applied to this population. If the return value of *condition* is `True`, the evolution of the population will be terminated. If *stopAll* is set to `True`, the evolution of all replicates of the simulator will be terminated. If this operator is allowed to write to an *output* or *outputExpr* (both default to ""), the generation number, preceeded with an optional *message* will be written to it.

`apply` (*pop*)
Apply to one population. It does not check if the operator is activated.

`clone` ()
Deep copy of a `terminateIf` terminator

2.14 Python operators

2.14.1 Class `pyOperator`

A python operator that directly operate a population. This operator accepts a function that can take the form of

- `func(pop)` when `stage=PreMating` or `PostMating`, without setting `param`;
- `func(pop, param)` when `stage=PreMating` or `PostMating`, with `param`;

- `func(pop, off, dad, mom)` when `stage=DuringMating` and `passOffspringOnly=False`, without setting `param`;
- `func(off)` when `stage=DuringMating` and `passOffspringOnly=True`, and without setting `param`;
- `func(pop, off, dad, mom, param)` when `stage=DuringMating` and `passOffspringOnly=False`, with `param`;
- `func(off, param)` when `stage=DuringMating` and `passOffspringOnly=True`, with `param`.

For `Pre-` and `PostMating` usages, a population and an optional parameter is passed to the given function. For `DuringMating` usages, population, offspring, its parents and an optional parameter are passed to the given function. Arbitrary operations can be applied to the population and offspring (if `stage=DuringMating`).

class `pyOperator` (*func, param=None, stage=PostMating, formOffGenotype=False, passOffspringOnly=False, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Python operator, using a function that accepts a population object.

func: A Python function. Its form is determined by other parameters.

param: Any Python object that will be passed to `func` after `pop` parameter. Multiple parameters can be passed as a tuple.

formOffGenotype: This option tells the mating scheme this operator will set the genotype of offspring (valid only for `stage=DuringMating`). By default (`formOffGenotype=False`), a mating scheme will set the genotype of offspring before it is passed to the given Python function. Otherwise, a 'blank' offspring will be passed.

passOffspringOnly: If `True`, `pyOperator` will expect a function of form `func(off [,param])`, instead of `func(pop, off, dad, mom [, param])` which is used when `passOffspringOnly` is `False`. Because many during-mating `pyOperator` only need access to offspring, this will improve efficiency. Default to `False`.

Note

- Output to `output` or `outputExpr` is not supported. That is to say, you have to open/close/append to files explicitly in the Python function. Because files specified by `output` or `outputExpr` are controlled (opened/closed) by simulators, they should not be manipulated in a `pyOperator` operator.
- This operator can be applied `Pre-`, `During-` or `Post- Mating` and is applied `PostMating` by default. For example, if you would like to examine the fitness values set by a selector, a `PreMating` Python operator should be used.

`apply`(*pop*)

Apply the `pyOperator` operator to one population

`clone`()

Deep copy of a `pyOperator` operator

2.14.2 Class `pyIndOperator`

Individual operator This operator is similar to a `pyOperator` but works at the individual level. It expects a function that accepts an individual, optional genotype at certain loci, and an optional parameter. When it is applied, it passes each individual to this function. When `infoFields` is given, this function should return an array to fill these `infoFields`. Otherwise, `True` or `False` is expected. More specifically, `func` can be

- `func(ind)` when neither `loci` nor `param` is given.
- `func(ind, genotype)` when `loci` is given.

- `func(ind, param)` when `param` is given.
- `func(ind, genotype, param)` when both `loci` and `param` are given.

class `pyIndOperator`(*func, loci=[], param=None, stage=PostMating, formOffGenotype=False, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

A `Pre-` or `PostMating` Python operator that apply a function to each individual

func: A Python function that accepts an individual and optional genotype and parameters.

param: Any Python object that will be passed to `func` after `pop` parameter. Multiple parameters can be passed as a tuple.

infoFields: If given, `func` is expected to return an array of the same length and fill these `infoFields` of an individual.

`apply`(*pop*)

Apply the `pyIndOperator` operator to one population

`clone`()

Deep copy of a `pyIndOperator` operator

2.15 Miscellaneous

2.15.1 Class `ifElse`

Conditional operator This operator accepts

- an expression that will be evaluated when this operator is applied.
- an operator that will be applied if the expression is `True` (default to null).
- an operator that will be applied if the expression is `False` (default to null).

When this operator is applied to a population, it will evaluate the expression and depending on its value, apply the supplied operator. Note that the `begin`, `end`, `step`, and `at` parameters of `ifOp` and `elseOp` will be ignored. For example, you can mimic the `at` parameter of an operator by `ifElse('rep in [2,5,9]' operator)`. The real use of this machanism is to monitor the population statistics and act accordingly.

class `ifElse`(*cond, ifOp=None, elseOp=None, output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a conditional operator

cond: Expression that will be treated as a boolean variable

ifOp: An operator that will be applied when `cond` is `True`

elseOp: An operator that will be applied when `cond` is `False`

`apply`(*pop*)

Apply the `ifElse` operator to one population

`clone`()

Deep copy of an `ifElse` operator

2.15.2 Class `turnOnDebug` (Function form: `TurnOnDebug`)

Set debug on Turn on debug. There are several ways to turn on debug information for non-optimized modules, namely

- set environment variable `SIMUDEBUG`.
- use `simuOpt.setOptions(debug)` function.
- use `TurnOnDebug` or `TurnOnDebugByName` function.
- use this `turnOnDebug` operator

The advantage of using this operator is that you can turn on debug at given generations.

class `turnOnDebug`(*code, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a `turnOnDebug` operator

`apply`(*pop*)
Apply the `turnOnDebug` operator to one population

`clone`()
Deep copy of a `turnOnDebug` operator

2.15.3 Class `turnOffDebug` (Function form: `TurnOffDebug`)

Set debug off Turn off debug.

class `turnOffDebug`(*code, stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)
Create a `turnOffDebug` operator

`apply`(*pop*)
Apply the `turnOffDebug` operator to one population

`clone`()
Deep copy of a `turnOffDebug` operator

2.15.4 Class `noneOp`

None operator This operator does nothing.

class `noneOp`(*output=">", outputExpr="", stage=PostMating, begin=0, end=0, step=1, at=[], rep=[], infoFields=[]*)
Create a none operator

`apply`(*pop*)
Apply the `noneOp` operator to one population

`clone`()
Deep copy of a `noneOp` operator

2.15.5 Class `pause`

Pause a simulator This operator pauses the evolution of a simulator at given generations or at a key stroke, using `stopOnKeyStroke=True` option. Users can use `'q'` to stop an evolution. When a simulator is stopped, press any other key to resume the simulation or escape to a Python shell to examine the status of the simulation by pressing `'s'`.

There are two ways to use this operator, the first one is to pause the simulation at specified generations, using the usual operator parameters such as `at`. Another way is to pause a simulation with any key stroke, using the `stopOnKeyStroke` parameter. This feature is useful for a presentation or an interactive simulation. When `'s'` is pressed, this operator expose the current population to the main Python dictionary as variable `pop` and enter an interactive Python session. The way current population is exposed can be controlled by parameter `exposePop` and `popName`. This feature is useful when you want to examine the properties of a population during evolution.

class pause(*prompt=True, stopOnKeyStroke=False, exposePop=True, popName="pop", output=">", outputExpr="", stage=PostMating, begin=0, end=-1, step=1, at=[], rep=-1, infoFields=[]*)

Stop a simulation. Press `'q'` to exit or any other key to continue.

prompt: If `True` (default), print prompt message.

stopOnKeyStroke: If `True`, stop only when a key was pressed.

exposePop: Whether or not expose `pop` to user namespace, only useful when user choose `'s'` at pause. Default to `True`.

popName: By which name the population is exposed. Default to `pop`.

apply(*pop*)

Apply the `pause` operator to one population

clone()

Deep copy of a `pause` operator

2.15.6 Class `ticToc` (Function form: `TicToc`)

Timer operator This operator, when called, output the difference between current and the last called clock time. This can be used to estimate execution time of each generation. Similar information can also be obtained from `turnOnDebug(DBG_PROFILE)`, but this operator has the advantage of measuring the duration between several generations by setting `step` parameter.

class ticToc(*output=">", outputExpr="", stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a timer

apply(*pop*)

Apply the `ticToc` operator to one population

clone()

Deep copy of a `ticToc` operator

2.15.7 Class `setAncestralDepth`

Set ancestral depth This operator set the number of ancestral generations to keep in a population. It is usually called like `setAncestral(at=[-2])` to start recording ancestral generations to a population at the end of the evolution. This is useful when constructing pedigree trees from a population.

class setAncestralDepth(*depth, output=">", outputExpr="", stage=PreMating, begin=0, end=-1, step=1, at=[], rep=[], infoFields=[]*)

Create a `setAncestralDepth` operator

apply(*pop*)

Apply the `setAncestralDepth` operator to one population

clone()

Deep copy of a `setAncestralDepth` operator

CHAPTER

THREE

Global and Python Utility functions

3.1 Global functions

AlleleType()
Return the allele type of the current module. Can be `binary`, `short`, or `long`.

Limits()
Print out system limits

ListAllRNG()
List the names of all available random number generators

ListDebugCode()
List all debug codes

LoadPopulation(*file*)
Load a population from a file.

LoadSimulator(*file, matingScheme*)
Load a simulator from a file with the specified mating scheme. The file format is by default determined by file extension (`format="auto"`). Otherwise, `format` can be one of `txt`, `bin`, or `xml`.

MaxAllele()
Return the maximum allowed allele state of the current simuPOP module, which is `1` for binary modules, `255` for short modules and `65535` for long modules.

MergePopulations(*pops, newSubPopSizes=[], keepAncestralPops=-1*)
Merge several populations with the same genotypic structure and create a new population

MergePopulationsByLoci(*pops, newNumLoci=[], newLociPos=[], byChromosome=False*)
Merge several populations of the same size by loci and create a new population

ModuleCompiler()
Return the compiler used to compile this simuPOP module

ModuleDate()
Return the date when this simuPOP module is compiled

ModulePlatForm()
Return the platform on which this simuPOP module is compiled

ModulePyVersion()
Return the Python version this simuPOP module is compiled for

Optimized()
Return `True` if this simuPOP module is optimized

SetRNG(*rng="", seed=0*)

Set random number generator. If `seed=0` (default), a random seed will be given. If `rng=""`, seed will be set to the current random number generator.

TurnOffDebug(*code=DBG_ALL*)
Turn off debug information. Default to turn off all debug codes. Only available in non-optimized modules.

TurnOnDebug(*code=DBG_ALL*)
Set debug codes. Default to turn on all debug codes. Only available in non-optimized modules.

rng()
Return the currently used random number generator

simuRev()
Return the revision number of this simuPOP module. Can be used to test if a feature is available.

simuVer()
Return the version of this simuPOP module

3.2 Utility Classes

3.2.1 Class `RNG`

Random number generator This random number generator class wraps around a number of random number generators from GNU Scientific Library. You can obtain and change system random number generator through the `rng()` function. Or create a separate random number generator and use it in your script.

class RNG(*rng=None, seed=0*)
RNG used by simuPOP.

max()
Maximum value of this RNG.

maxSeed()
Return the maximum allowed seed value

name()
Return RNG name

pvalChiSq(*chisq, df*)
Right hand side (single side) p-value for ChiSq value

randBinomial(*n, p*)
Binomial distribution B(n, p).

randExponential(*v*)
FIXME: No document

randGeometric(*p*)
Geometric distribution.

randGet()
Return a random number in the range of [0, 2, ... max()-1]

randInt(*n*)
Return a random number in the range of [0, 1, 2, ... n-1]

randMultinomial(*N, p, n*)
Multinomial distribution.

randMultinomialVal(*N, p*)
FIXME: No document

randNormal(*m, v*)
Normal distribution.

randPoisson(*p*)
Poisson distribution.

randUniform01()
Uniform distribution [0,1).

seed()
Return the seed of this RNG

setRNG(*rng=None, seed=0*)
Choose an random number generator, or set seed to the current RNG

rng: Name of the RNG. If rng is not given, environmental variable GSL_RNG_TYPE will be used if it is available. Otherwise, RNG`mt19937` will be used.

seed: Random seed. If not given, `/dev/urandom`, `/dev/random`, system time will be used, depending on availability, in that order. Note that windows system does not have `/dev` so system time is used.

setSeed(*seed*)
If seed is 0, method described in `setRNG` is used.

3.3 Utility Modules

Several utility modules are distributed with simuPOP. They provide important functions and extensions to simuPOP and serve as good examples on how simuPOP can be used.

Compared to simuPOP kernel functions, these utility functions are less tested, and are subject to more frequent changes. Please report to simuPOP mailing list if any function stops working.

3.3.1 Module `simuOpt`

Module simuOpt can be used to control which simuPOP module to load, and how it is loaded using function setOptions . It also provides a simple way to set simulation options, from user input, command line, configuration file or a parameter dialog. All you need to do is to define an option description list that lists all parameters in a given format, and call the getParam function.

This module, if loaded, pre-process the command line options. More specifically, it checks command line option:

-c configfile: read from a configuration file

--config configfile: the same as -c

--optimized: load optimized modules, unless setOption explicitly use non-optimized: modules.

-q: Do not display banner information when simuPOP is loaded

--quiet: the same as -q

--useTkinter: force the use of Tcl/Tk dialog even when wxPython is available. By: default, wxPython is used whenever possible.

--noDialog: do not use option dialog. If the options can not be obtained from: command line or configuraiton file, users will be asked to input them interactively.

Because these options are reserved, you can not use them in your simuPOP script.

Module Functions

getParam(*options=[], doc='', details='', noDialog=False, UnprocessedArgs=True, verbose=False, nCol=1*)
Get parameters from either:

- a Tcl/Tk based, or wxPython based parameter dialog (wxPython is used if it is available)
- command line argument
- configuration file specified by -c file (--config file), or
- prompt for user input

The option description list consists of dictionaries with some predefined keys. Each dictionary defines an option. Each option description item can have the following keys:

arg: short command line option name. 'h' checks the presence of argument -h . If an argument is expected, add a comma to the option name. For example, 'p:' matches command line option -p=100 or -p 100 .

longarg: long command line option name. 'help' checks the presence of: argument '--help' . 'mu=' matches command line option --mu=0.001 or -mu 0.001 .

label: The label of the input field in a parameter dialog, and as the prompt for: user input.

default: default value for this parameter. It is used to as the default value: in the parameter dialog, and as the option value when a user presses 'Enter' directly during interactive parameter input.

useDefault: use default value without asking, if the value can not be determined: from GUI, command line option or config file. This is useful for options that rarely need to be changed. Setting them to useDfault allows shorter command lines, and easy user input.

description: a long description of this parameter, will be put into the usage: information, which will be displayed with (-h , --help command line option, or help button in parameter dialog).

allowedTypes: acceptable types of this option. If allowedTypes is types.ListType: or types.TupleType and the user's input is a scalar, the input will be converted to a list automatically. If the conversion can not be done, this option will not be accepted.

validate: a function to validate the parameter. You can define your own functions: or use the ones defined in this module.

chooseOneOf: if specified, simuOpt will choose one from a list of values using a: listbox (Tk) or a combo box (wxPython) .

chooseFrom: if specified, simuOpt will choose one or more items from a list of: values using a listbox (tk) or a combo box (wxPython).

separator: if specified, a blue label will be used to separate groups of: parameters.

jump: it is used to skip some parameters when doing the interactive user input.: For example, getParam will skip the rest of the parameters if -h is specified if parameter -h has item 'jump':-1 which means jumping to the end. Another situation of using this value is when you have a hierarchical parameter set. For example, if mutation is on, specify mutation rate, otherwise proceed. The value of this option can be the absolute index or the longarg name of another option.

jumpIfFalse: The same as jump but jump if current parameter is False .

This function will first check command line argument. If the argument is available, use its value. Otherwise check if a config file is specified. If so, get the value from the config file. If both failed, prompt user to input a value. All input will be checked against types, if exists, an array of allowed types.

Parameters of this function are:

options: a list of option description dictionaries

doc: short description put to the top of parameter dialog

details: module help. Usually set to __doc__ .

noDialog: do not use a parameter dialog, used in batch mode. Default to False.

checkUnprocessedArgs: obsolete because unused args are always checked.

verbose: whether or not print detailed info

nCol: number of columns in the parameter dialog.

prettyOutput(*value, quoted=False, outer=True*)
Return a value in good format, the main purpose is to avoid [0.90000001, 0.2].

printConfig(*opt, param, out=<open file '<stdout>', mode 'w' at 0x2aaaaaad6198>*)
Print configuration.

opt: option description list

param: parameters returned from getParam()

out: output

requireRevision(*rev*)
Compare the revision of this simuPOP module with given revision. Raise an exception if current module is out of date.

saveConfig(*opt, file, param*)
Write a configuration file. This file can be later read with command line option -c or --config .

opt: the option description list

file: output file

param: parameters returned from getParam

setOptions(*optimized=None, mpi=None, chromMap=[], alleleType=None, quiet=None, debug=[]*)
set options before simuPOP is loaded to control which simuPOP module to load, and how the module should be loaded.

optimized: whether or not load optimized version of a module. If not set,: environmental variable SIMUOPTIMIZED, and commandline option --optimized will be used if available. If nothing is defined, standard version will be used.

mpi: obsolete.

chromMap: obsolete.

alleleType: 'binary', 'short', or 'long'. 'standard' can be used as 'short': for backward compatibility. If not set, environmental variable SIMUALLELETYPE will be used if available. if it is not defined, the short allele version will be used.

quiet: If True, supress banner information when simuPOP is loaded.

debug: a list of debug code (or string). If not set, environmental variable: SIMUDEBUG will be used if available.

usage(*options, before=''*)
Print usage information from the option description list. Used with -h (or --help) option, and in the parameter input dialog.

options: option description list.

before: optional information

valueAnd(*t1, t2*)
Return a function that returns true if passed option passes validator t1 and t2

valueBetween(*a, b*)
Return a function that returns true if passed option is between value a and b (a and b included)

valueEqual(*a*)
Return a function that returns true if passed option equals a

valueGE(*a*)
Return a function that returns true if passed option is greater than or equal to a

valueGT(*a*)
Return a function that returns true if passed option is greater than a

valueIsList()
Return a function that returns true if passed option is a list (or tuple)

valueIsNum()
Return a function that returns true if passed option is a number (int, long or float)

valueLE(*a*)
Return a function that returns true if passed option is less than or equal to a

valueLT(*a*)
Return a function that returns true if passed option is less than a

valueListOf(*t*)
Return a function that returns true if passed option val is a list of type t. If t is a function (validator), check if all v in val pass t(v)

valueNot(*t*)
Return a function that returns true if passed option does not passes validator t

valueNotEqual(*a*)
Return a function that returns true if passed option does not equal a

valueOneOf(*t*)
Return a function that returns true if passed option is one of the values list in t

valueOr(*t1, t2*)
Return a function that returns true if passed option passes validator t1 or t2

valueTrueFalse()
Return a function that returns true if passed option is True or False

valueValidDir()
Return a function that returns true if passed option val if a valid directory

valueValidFile()
Return a function that returns true if passed option val if a valid file

3.3.2 Module `simuUtil`

This module provides some commonly used operators and format conversion utilities.

Module Functions

CaseControl_ChiSq(*pop, sampleSize, penetrance=None*)
Draw affected sibpair sample from pop, run TDT using GENEHUNTER

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)
- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

pene: penetrance function, if not given (None), existing affection status will be used.

sampleSize: total sample size N. N/4 is the number of families to ascertain.

keep_temp: if True, do not remove sample data. Default to False.

ChiSq_test(*pop*)
perform case control test

pop: loaded population, or population file in simuPOP format. This function assumes that pop has two subpopulations, cases and controls, and have 0 as wildtype and 1 as disease allele. pop can also be an loaded

population object.

Return value: A list of p-value at each locus.

Note: this function requires rpy module.

ConstSize (*size, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0*)
The population size is constant, but will split into numSubPop subpopulations at generation split

ExponentialExpansion (*initSize, endSize, end, burnin=0, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0*)
Exponentially expand population size from intiSize to endSize after burnin, split the population at generation split.

InstantExpansion (*initSize, endSize, end, burnin=0, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0*)
Instaneously expand population size from intiSize to endSize after burnin, split the population at generation split.

LOD_gh (*file, gh='gh'*)
Analyze data using the linkage method of genehunter. Note that this function may not work under platforms other than linux, and may not work with your version of genehunter. As a matter of fact, it is almost unrelated to simuPOP and is provided only as an example how to use python to analyze data.

Parameters: file: file to analyze. This function will look for file.dat and file.pre in linkage format.

loci: a list of loci at which p-value will be returned. If the list is empty, all p-values are returned.

gh: name (or full path) of genehunter executable. Default to 'gh'

Return value: A list (for each chromosome) of list (for each locus) of p-values.

LOD_merlin (*file, merlin='merlin'*)
run multi-point non-parametric linkage analysis using merlin

LargePeds_Reg_merlin (*pop, sampleSize, qtrait=None, infoField='qtrait', merlin='merlin-regress', keep_temp=False*)
Draw affected sibpair sample from pop, run TDT using GENEHUNTER

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)
- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

qtrait: a function to calculate quantitative trait

infoField: information field to store quantitative trait. Default to 'qtrait'

sampleSize: total sample size N. N/4 is the number of families to ascertain.

merlin: executable name of merlin, full path name can be given.

keep_temp: if True, do not remove sample data. Default to False.

LargePeds_VC_merlin (*pop, sampleSize, qtrait=None, infoField='qtrait', merlin='merlin', keep_temp=False*)
Draw affected sibpair sample from pop, run TDT using GENEHUNTER

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)
- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

qtrait: a function to calculate quantitative trait

infoField: information field to store quantitative trait. Default to 'qtrait'

sampleSize: total sample size N. N/4 is the number of families to ascertain.

merlin: executable name of merlin, full path name can be given.

keep_temp: if True, do not remove sample data. Default to False.

LinearExpansion(*initSize, endSize, end, burnin=0, split=0, numSubPop=1, bottleneckGen=-1, bottleneckSize=0*)

Linearly expand population size from intiSize to endSize after burnin, split the population at generation split.

ListVars(*var, level=-1, name='', subPop=True, useWxPython=True*)

list a variable in tree format, either in text format or in a: wxPython window.

var: any variable to be viewed. Can be a dw object returned by dvars() function

level: level of display.

name: only view certain variable

subPop: whether or not display info in subPop

useWxPython: if True, use terminal output even if wxPython is available.

LoadFstat(*file, loci=[]*)

load population from fstat file 'file' since fstat does not have chromosome structure an additional parameter can be given

MigrIslandRates(*r, n*)

migration rate matrix

```
x m/(n-1) m/(n-1) ....
m/(n-1) x .............
.....
.... m/(n-1) m/(n-1) x
```

where x = 1-m

MigrSteppingStoneRates(*r, n, circular=False*)

migration rate matrix, circular stepping stone model (X=1-m)

```
X   m/2               m/2
m/2 X   m/2           0
0   m/2 x   m/2 ......0
...
m/2 0 ....        m/2  X
```

or non-circular

```
X   m/2               m/2
m/2 X   m/2           0
0   m/2 X   m/2 ......0
...
...               m   X
```

QtraitSibs_Reg_merlin(*pop, sampleSize, qtrait=None, infoField='qtrait', merlin='merlin-regress', keep_temp=False*)

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)

- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

qtrait: a function to calculate quantitative trait

infoField: information field to store quantitative trait. Default to 'qtrait'

sampleSize: total sample size N. N/4 is the number of families to ascertain.

merlin: executable name of merlin, full path name can be given.

keep_temp: if True, do not remove sample data. Default to False.

QtraitSibs_VC_merlin (*pop, sampleSize, qtrait=None, infoField='qtrait', merlin='merlin', keep_temp=False*)

Draw affected sibpair sample from pop, run TDT using GENEHUNTER

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)
- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

qtrait: a function to calculate quantitative trait

infoField: information field to store quantitative trait. Default to 'qtrait'

sampleSize: total sample size N. N/4 is the number of families to ascertain.

merlin: executable name of merlin, full path name can be given.

keep_temp: if True, do not remove sample data. Default to False.

Regression_merlin (*file, merlin='merlin-regress'*)

run merlin regression method

SaveCSV (*pop, output='', outputExpr='', fields=['sex', 'affection'], loci=[], combine=None, shift=1, **kwargs*)

save file in CSV format

fileds: information fields, 'sex' and 'affection' are special fields that is treated differently.

genotype: list of loci to output, default to all.

combine: how to combine the markers. Default to None. A function can be specified, that takes the form:

```
def func(markers):
    return markers[0]+markers[1]
```

shift: since alleles in simuPOP is 0-based, shift=1 is usually needed to output alleles starting from allele 1. This parameter is ignored if combine is used.

SaveFstat (*pop, output='', outputExpr='', maxAllele=0, loci=[], shift=1, combine=None*)

save file in FSTAT format

SaveLinkage (*pop, output='', outputExpr='', loci=[], shift=1, combine=None, fields=[], recombination=1.0000000000000001e-05, penetrance=[0, 0.25, 0.5], affectionCode=['1', '2'], pre=True, daf=0.001*)

save population in Linkage format. Currently only support affected sibpairs sampled with affectedSibpairSample operator.

pop: population to be saved. Must have ancestralDepth 1. paired individuals are sibs. Parental population are corresponding parents. If pop is a filename, it will be loaded.

output: Output.dat and output.ped will be the data and pedigree file. You may need to rename them to be analyzed by LINKAGE. This allows saving multiple files.

outputExpr: expression version of output.

affectionCode: default to '1': unaffected, '2': affected

pre: True. pedigree format to be fed to makeped. Non-pre format it is likely to be wrong now for non-sibpair families.

Note: the first child is always the proband.

SaveMerlinDatFile(*pop, output='', outputExpr='', loci=[], fields=[], outputAffection=False*)
Output a .dat file readable by merlin

SaveMerlinMapFile(*pop, output='', outputExpr='', loci=[]*)
Output a .map file readable by merlin

SaveMerlinPedFile(*pop, output='', outputExpr='', loci=[], fields=[], header=False, outputAffection=False, affectionCode=['U', 'A'], combine=None, shift=1, **kwargs*)
Output a .ped file readable by merlin

SaveQTDT(*pop, output='', outputExpr='', loci=[], header=False, affectionCode=['U', 'A'], fields=[], combine=None, shift=1, **kwargs*)
save population in Merlin/QTDT format. The population must have pedindex, father_idx and mother_idx information fields.

pop: population to be saved. If pop is a filename, it will be loaded.

output: base filename.

outputExpr: expression for base filename, will be evaluated in pop's local namespace.

affectionCode: code for unaffected and affected. '1', '2' are default, but 'U', and 'A' or others can be specified.

loci: loci to output

header: whether or not put head line in the ped file.

fields: information fields to output

combine: an optional function to combine two alleles of a diploid individual.

shift: if combine is not given, output two alleles directly, adding this value (default to 1).

SaveSolarFrqFile(*pop, output='', outputExpr='', loci=[], calcFreq=True*)
Output a frequency file, in a format readable by solar calcFreq

Unexpected indentation.

whether or not calculate allele frequency

Sibpair_LOD_gh(*pop, sampleSize, penetrance=None, recRate=None, daf=None, gh='gh', keep_temp=False*)
Draw affected sibpair sample from pop, run Linkage analysis using GENEHUNTER

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)
- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

pene: penetrance function, if not given (None), existing affection status will be used.

sampleSize: total sample size N. N/4 is the number of families to ascertain.

recRate: recombination rate, used in the Linkage file. If not given, pop.dvars().recRate[0] will be used. If there is no such variable, 0.0001 is used.

daf: disease allele frequency. This is needed for the linkage format but I am not sure if it is used by TDT.

gh: executable name of genehunter, full path name can be given.

keep_temp: if True, do not remove sample data. Default to False.

Sibpair_LOD_merlin (*pop, sampleSize, penetrance=None, merlin='merlin', keep_temp=False*)
Draw affected sibpair sample from pop, run multi-point linkage analysis using merlin

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)
- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

pene: penetrance function, if not given (None), existing affection status will be used.

sampleSize: total sample size N. N/4 is the number of families to ascertain.

merlin: executable name of merlin, full path name can be given.

keep_temp: if True, do not remove sample data. Default to False.

Sibpair_TDT_gh (*pop, sampleSize, penetrance=None, recRate=None, daf=None, gh='gh', keep_temp=False*)
Draw affected sibpair sample from pop, run TDT using GENEHUNTER

pop: simuPOP population. It can be a string if path to a file is given. This population must

- have at least one ancestral generation (parental generation)
- have a variable DSL (pop.dvars().DSL) indicating the Disease susceptibility loci. These DSL will be removed from the samples.
- has only binary alleles

pene: penetrance function, if not given (None), existing affection status will be used.

sampleSize: total sample size N. N/4 is the number of families to ascertain.

recRate: recombination rate, used in the Linkage file. If not given, pop.dvars().recRate[0] will be used. If there is no such variable, 0.0001 is used.

daf: disease allele frequency. This is needed for the linkage format but I am not sure if it is used by TDT.

gh: executable name of genehunter, full path name can be given.

keep_temp: if True, do not remove sample data. Default to False.

TDT_gh (*file, gh='gh'*)
Analyze data using genehunter/TDT. Note that this function may not work under platforms other than linux, and may not work with your version of genehunter. As a matter of fact, it is almost unrelated to simuPOP and is provided only as an example how to use python to analyze data.

Parameters: *file:* file to analyze. This function will look for file.dat and file.pre in linkage format.

loci: a list of loci at which p-value will be returned. If the list is empty, all p-values are returned.

gh: name (or full path) of genehunter executable. Default to 'gh'

Return value: A list (for each chromosome) of list (for each locus) of p-values.

VC_merlin (*file, merlin='merlin'*)
run variance component method

file: file.ped, file.dat, file.map and file,mdl are expected. file can contain directory name.

saveFstat (*output='', outputExpr='', **kwargs*)
operator version of the function SaveFstat

saveLinkage (*output='', outputExpr='', **kwargs*)
An operator to save population in linkage format

simuProgress (*self, message, totalCount, progressChar='.', block=2, done=' Done.
n'*)
This class defines a very simple text based progress bar. It will display a character (default to '.') for

each change of progress (default to 2%), and a number (1, 2, ..., 9) for each 10% of progress, and print a message (default to 'Done.

Block quote ends without a blank line; unexpected unindent.

') when the job is finished.

This progress is used as follows:

progress = simuProgress(500) for i in range(500):
Unexpected indentation.

progress.update(i+1)

Block quote ends without a blank line; unexpected unindent.
if you would like to make sure the done message is displayed. progress.done()totalCount

Block quote ends without a blank line; unexpected unindent.

Total expected steps.

progressChar: Character to be displayed for each progress.

block: display progress at which interval (in terms of percentage)?

done: Message displayed when the job is finished.

3.3.3 Module `simuRPy`

This module helps the use of `rpy` package with simuPOP. It defines an operator `varPlotter` that can be used to plot population expressions when `rpy` is installed.

Module Functions

`rmatrix`(*mat*)
Convert a Python 2d list to r matrix format that can be passed to functions like image directly.

`varPlotter`(*self, expr, history=True, varDim=1, numRep=1, win=0, ylim=[0, 0], update=1, title='', xlab='generation', ylab='', axes=True, lty=[], col=[], mfrow=[1, 1], separate=False, byRep=False, byVal=False, plotType='plot', level=20, saveAs='', leaveOpen=True, dev='', width=0, height=0, *args, **kwargs*)
This class defines a Python operator that uses R to plot a simuPOP express. During the evolution, this express is evaluated in each replicate's local namespace. How this expression is plotted depends on the dimension of the return value (if a sequence is returned), number of replicates, whether or not historical values (collected over several generations) are plotted, and plot type (lines or images).

The default behavior of this operator is to plot the history of an expression. For example, when operator

varPlotter(var='expr')

is used in simulator::evolve, the value of `expr` will be recorded each time when this operator is applied. A line will be draw in a figure with x-axis being the generation number. Parameters `ylim` can be used to specify the range of y-axis.

If the return value of expression `expr` is a sequence (tuple or list), parameter `varDim` has to be used to indicate the dimension of this expression. For example,

varPlotter(var='expr', varDim=3)

will plot three lines, corresponding to the histories of each item in the array.

If the expression returns a number and there are several replicates, parameter `numRep` ' should be used. In this case, each line will correspond to a replicate.

If the expression returns a vector and there are several replicates, several subplots will be used. Parameter `byRep` or `byVar` should be used to tell `varPlotter` whether the subplots should be divided by replicate or by variable. For example,

varPlotter(var='expr', varDim=8, numRep=5, byRep=1)

will use an appropriate layout for your subplots, which is, in this case, 2x3 for 5 replicates. Each subplot will have 8 lines. If byVal is `True`, there will be 3x3 subplots for 8 items in an array, and each subplot will have 5 lines. Note that `byRep` or `byVal` can also be used when there is only one replicate or if the dimension of the expression is one.

When `history=False`, histories of each variable will be discarded so the figure will always plot the current value of the expression.expr

Unexpected indentation.

expression that will be evaluate at each replicate's local namespace when the operator is applied.

history: whether or not record and plot the history of an expression. Default to True.

varDim: If the return value of `expr` is a sequence, `varDim` should be set to the length of this sequence. Default to 1.

numRep: Number of replicates of the simulator. Default to 1.

win: Window of generations. I.e., how many generations to keep in a figure. This is useful when you want to keep track of only recent changes of an expression. The default value is 0, which will keep all histories.

ylim: The range of y-axis.

update: Update figure after update generations. This is used when you do not want to update the figure every time when this operator is applied.

title, xlab, ylab: Title, label at x and y axes of your figure(s). xtitle is defaulted to 'generation'.

axes: Whether or not plot axes. Default to `True`.

lty: A list of line type for each line in the figure.

col: A list of colors for each line in the figure.

level: level of image colors (default to 20).

saveAs: save figures in files saveAs#gen.eps. For example, if saveAs='demo', you will get files demo1.eps, demo2.eps etc.

separate: plot data lines in separate panels.

image: use R image function to plot image, instead of lines.

leaveOpen: whether or not leave the plot open when plotting is done. Default to True.

3.3.4 Module `hapMapUtil`

Utility functions to manipulate HapMap data. These functions are provided as examples on how to load and evolve the HapMap dataset. They tend to change frequently so do not call these functions directly. It is recommended that you copy these function to your script when you need to use them.

Module Functions

`evolveHapMap`(*pop, endingSize, gen, migr=<simuPOP_std.noneOp; proxy of <Swig Object of type 'simuPOP::noneOp *' at 0x18fd3990> >, expand='exponential', mergeAt=10000, initMultiple=1, recIntensity=0.01, mutRate=9.9999999999999995e-08, step=10, keepParents=False, numOffspring=1, recordAncestry=False*)

Evolve and expand the hapmap population

gen: total evolution generation

initMultiple: copy each individual initMultiple times, to avoid: rapid loss of genotype variation when population size is small.

endingSize: ending poplation size

expand: expanding method, can be linear or exponential

mergeAt: when to merge population?

gen: generations to evolve

migr: a migrator to be used.

recIntensity: recombination intensity

mutRate: mutation rate

step: step at which to display statistics

keepParents: whether or not keep parental generations

numOffspring: number of offspring at the last generation

recordAncestry: whether or not calculate ancestry to an information field: ancestry. Only usable with two hapmap populations.

getMarkersFromName(*HapMap_dir, names, chroms=[], hapmap_pops=[], minDiffAF=0, numMarkers=[]*)

Get population from marker names. This function: returns a tuple with a population with found markers and names of markers that can not be located in the HapMap data. The returned population has three subpopulations, corresponding to CEU, YRI and JPT+CHB HapMap populations.

HapMap_dir: where HapMap data in simuPOP format is stored. The files: should have been prepared by scripts/loadHapMap.py.

names: names of markers. It can either be a stright list of names, or: a dictionary of names categorized by chromosome number.

chroms: a list of chromosomes to look in. If empty, all 22 autosomes: will be tried. Chromosome index starts from 1. (1, ..., 22).

hapmap_pops: hapmap populations to load, can be a list of 'CEU', 'YRI': or 'JPT+CHB', or a list of 0, 1, 2. If empty (default), all three populations will be loaded.

minDiffAF: minimal allele frequency difference between hapmap populations.: If three subpopulations are loaded, use the maximal of three pair-wise allele frequency differences for comparison. This option is ignored if hapmap_pops has length one.

numMarkers: number of markers to use for each chromosome. Must have: the same length as chroms.

getMarkersFromRange(*HapMap_dir, hapmap_pops, chrom, startPos, endPos, maxNum, minAF=0, minDiffAF=0, minDist=0, maxDist=0*)

Get a population with markers from given range

HapMap_dir: where HapMap data in simuPOP format is stored. The files: should have been prepared by scripts/loadHapMap.py.

hapmap_pops: HapMap populations to load. It can be a list of 'CEU', 'YRI': or 'JPT+CHB', or a list of 0, 1, 2. If empty, all hapmap populations will be loaded.

chrom: chromosome number (1-based index)

startPos: starting position (in cM)

endPos: ending position (in cM). If 0, ignore this parameter.

maxNum: maximum number of markers to get. If 0, ignore this parameter.

minAF: minimal minor allele frequency

minDiffAf: minimal allele frequency between HapMap populations.

minDist: minimal distance between two adjacent markers, in cM

maxDist: maximum distance. If exceed, try to pick up a marker ASAP.

sample1DSL(*pop, DSL, DA, pene, name, sampleSize*)

Sample from the final population, using a single lcous penetrance model.

DSL: disease locus

DA: disease allele

pene: penetrance

name: name of directory to save (it must exist)

sampleSize: sample size, in this case, sampleSize/4 is the number of families

sample2DSL(*pop, DSL, pene, name, size*)

Sample from the final population, using a two locus penetrance model

DSL: disease loci (two locus)

pene: penetrance value, assuming a two-locus model

name: name to save sample

size: sample size

INDEX