
simuPOP

February 08, 2019

Contents

1	Front Matter	1
2	Introduction	3
2.1	What is simuPOP?	3
2.2	An overview of simuPOP concepts	4
2.3	Features	6
2.4	License, Distribution and Installation	7
2.5	How to read this user's guide	8
2.6	Other help sources	9
3	Loading and running simuPOP	11
3.1	Pythonic issues	11
3.2	Loading simuPOP modules	16
3.3	Online help system	18
3.4	Debug-related functions and operators *	19
3.5	Random number generator *	21
4	Individuals and Populations	23
4.1	Genotypic structure	23
4.2	Individual	29
4.3	Population	32
5	simuPOP Operators	51
5.1	Introduction to operators	51
5.2	Initialization	59
5.3	Expressions and statements	63
5.4	Demographic changes	67
5.5	Genotype transmitters	78
5.6	Mutation	83
5.7	Penetrance	98
5.8	Quantitative trait	103
5.9	Natural Selection	104
5.10	Tagging operators	117
5.11	Statistics calculation (operator <code>Stat</code>)	124
5.12	Conditional operators	145
5.13	Miscellaneous operators	148
5.14	Hybrid and Python operators	152

6	Evolving populations	157
6.1	Mating Schemes	157
6.2	Simulator	178
6.3	Non-random and customized mating schemes *	183
6.4	Age structured populations with overlapping generations **	193
6.5	Tracing allelic lineage *	196
6.6	Pedigrees	198
6.7	Evolve a population following a specified pedigree structure **	204
6.8	Simulation of mitochondrial DNAs (mtDNAs) *	208
7	Utility Modules	211
7.1	Module <code>simuOpt</code> (function <code>simuOpt.setOptions</code>)	211
7.2	Module <code>simuPOP.utils</code>	211
7.3	Module <code>simuPOP.demography</code>	223
7.4	Module <code>simuPOP.sampling</code>	240
7.5	Module <code>simuPOP.gsl</code>	245
8	A real world example	247
8.1	Simulation scenario	247
8.2	Demographic model	247
8.3	Mutation and selection models	249
8.4	Output statistics	249
8.5	Initialize and evolve the population	251
8.6	Option handling	252
9	Front Matter	259
10	simuPOP Components	261
10.1	Individual, Population, pedigree and Simulator	261
10.2	Virtual splitters	276
10.3	Mating Schemes	280
10.4	Pre-defined mating schemes	286
10.5	Utility Classes	289
10.6	Global functions	291
11	Operator References	295
11.1	Base class for all operators	295
11.2	Initialization	296
11.3	Expression and Statements	298
11.4	Demographic models	300
11.5	Genotype transmitters	303
11.6	Mutation	307
11.7	Penetrance	311
11.8	Quantitative Trait	314
11.9	Natural selection	315
11.10	Tagging operators	318
11.11	Statistics Calculation	321
11.12	Conditional operators	329
11.13	The Python operator	330
11.14	Miscellaneous operators	331
11.15	Function form of operators	333
12	Utility Modules	337
12.1	Module <code>simuOpt</code>	337
12.2	Module <code>simuPOP.utils</code>	338

12.3	Module <code>simuPOP.demography</code>	347
12.4	Module <code>simuPOP.sampling</code>	354
12.5	Module <code>simuPOP.gsl</code>	360
Python Module Index		363

Abstract

simuPOP is a general-purpose individual-based 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. In contrast to competing applications that use command-line options or configuration files to direct the execution of a limited number of predefined evolutionary scenarios, users of simuPOP's scripting interface could make use of many of its unique features, such as customized chromosome types, arbitrary nonrandom mating schemes, virtual subpopulations, information fields and Python operators, to construct and study almost arbitrarily complex evolutionary scenarios.

simuPOP is provided as a number of Python modules, which consist of a large number of Python objects and functions, including population, mating schemes, operators (objects that manipulate populations) and simulators to coordinate the evolutionary processes. It is the users' responsibility to write a Python script to glue these pieces together and form a simulation. At a more user-friendly level, an increasing number of functions and scripts contributed by simuPOP users is available in the online simuPOP cookbook. They provide useful functions for different applications (e.g. load and manipulate HapMap samples, import and export files from another application) and allow users who are unfamiliar with simuPOP to perform a large number of simulations ranging from basic population genetics models to generating datasets under complex evolutionary scenarios.

This user's guide shows you how to install and use simuPOP using a large number of examples. It describes all important concepts and features of simuPOP and demonstrates how to use them in a simuPOP script. Although the new Python 3.x releases are incompatible with Python 2.x, examples in this book are written in a style that is compatible with both versions of Python. For a complete and detailed description about all simuPOP functions and classes, please refer to the *simuPOP Reference Manual*. All resources, including a pdf version of this guide and a mailing list can be found at the simuPOP homepage <http://simupop.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.

2.1 What is simuPOP?

simuPOP is a **general-purpose individual-based forward-time population genetics simulation environment** based on Python, a dynamic object-oriented programming language that has been widely used in biological studies. More specifically,

- simuPOP is a **population genetics simulator** that simulates the evolution of populations. It uses a discrete generation model although overlapping generations could be simulated using nonrandom mating schemes.
- simuPOP explicitly models **populations with individuals** who have their own genotype, sex, and auxiliary information such as age. The evolution of a population is modeled by populating an offspring population from parents in the parental population.
- 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.
- simuPOP is a **general-purpose** simulator that is designed to simulate arbitrary evolutionary processes. In contrast to competing applications that use command-line options or configuration files to direct the execution of a limited number of predefined evolutionary scenarios, users of simuPOP's scripting interface could make use of many of its unique features, such as customized chromosome types, arbitrary nonrandom mating schemes, virtual subpopulations, information fields and Python operators, to construct and study almost arbitrarily complex evolutionary scenarios. In addition, because simuPOP provides a large number of functions to manipulate populations, it can be used as an data manipulation and analysis tool.

simuPOP is provided as a number of Python modules, which consist of a large number of Python objects and functions, including Population, mating schemes, operators (objects that manipulate populations) and simulators to coordinate the evolutionary processes. It is the users' responsibility to write a Python script to glue these pieces together and form a simulation. At a more user-friendly level, an increasing number of functions and scripts contributed by simuPOP users is available in the online simuPOP cookbook (<http://simupop.sourceforge.net/cookbook>). They provide useful functions for different applications (e.g. load and manipulate HapMap samples, import and export files from another application) and allow users who are unfamiliar with simuPOP to perform a large number of simulations ranging from basic population genetics models to generating datasets under complex evolutionary scenarios.

2.2 An overview of simuPOP concepts

A simuPOP **population** consists of **individuals** of the same **genotype structure**, which includes properties such as number of homologous sets of chromosomes (ploidy), number of chromosomes, and names and locations of markers on each chromosome. In addition to basic information such as genotypes and sex, individuals can have arbitrary auxiliary values as **information fields**. Individuals in a population can be divided into **subpopulations** that can be further grouped into **virtual subpopulations** according to individual properties such as sex, affection status, or arbitrary auxiliary information such as age. Whereas subpopulations define boundaries of individuals that restrict the flow of individuals and their genotypes (mating happens within subpopulations), virtual subpopulations are groups of individuals who share the same properties, with membership of individuals change easily with change of individual properties.

Figure: *A life cycle of an evolutionary process*

Illustration of the discrete-generation evolutionary model used by simuPOP.

Operators are Python objects that act on a population. They can be applied to a population before or after mating during a life cycle of an evolutionary process (Figure [fig_life_cycle](#)), or to parents and offspring during the production of each offspring. Arbitrary numbers of operators can be applied to an evolving population.

A simuPOP **mating scheme** is responsible for choosing parent or parents from a parental (virtual) subpopulation and for populating an offspring subpopulation. simuPOP provides a number of pre-defined **homogeneous mating schemes**, such as random, monogamous or polygamous mating, selfing, and haplodiploid mating in hymenoptera. More complicated nonrandom mating schemes such as mating in age-structured populations can be constructed using **heterogeneous mating schemes**, which applies multiple homogeneous mating schemes to different (virtual) subpopulations.

simuPOP evolves a population generation by generation, following the evolutionary cycle depicted in Figure [fig_life_cycle](#). Briefly speaking, a number of **operators** such as a *KAlleleMutator* are applied to a population before a mating scheme repeatedly chooses a parent or parents to produce offspring. **During-mating operators** such as *Recombinator* can be applied by a mating scheme to transmit parental genotype to offspring. After an offspring population is populated, other **operators** can be applied, for example, to calculate and output population statistics. The offspring population will then become the parental population of the next evolutionary cycle. Many simuPOP operators can be applied in different stages so the type of an operator is determined by the stage at which it is applied. Several populations, or replicates of a single population, could form a **simulator** and evolve together.

Example: *A simple example*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=0.01)),
...     postOps=[
...         sim.Stat(LD=[0, 1], step=10),
...         sim.PyEval(r"%0.2f\n" % LD[0][1], step=10),
...     ],
...     gen=100
... )
0.25
0.23
0.20
0.20
0.18
```

(continues on next page)



Users/bpeng1/simuPOP/simuPOP/doc/figures/evolve.png

(continued from previous page)

```
0.15
0.12
0.10
0.10
0.11
100

now exiting runScriptInteractively...
```

[Download simpleExample.py](#)

Some of these concepts are demonstrated in Example *simple_example*, where a standard diploid Wright-Fisher model with recombination is simulated. The first line imports the standard simuPOP module. The second line creates a diploid population with 1000 individuals, each having one chromosome with two loci. The `evolve()` function evolves the population using a random mating scheme and four operators.

Operators *InitSex* and *InitGenotype* are applied at the beginning of the evolutionary process. Operator *InitSex* initializes individual sex randomly and *InitGenotype* initializes all individuals with the same genotype 12/21. The populations are then evolved for 100 generations. A random mating scheme is used to generate offspring. Instead of using the default Mendelian genotype transmitter, a *Recombinator* (during-mating operator) is used to recombine parental chromosomes with the given recombination rate 0.01 during the generation of offspring. The other operators are applied to the offspring generation (post-mating) at every 10 generations (parameter `step`). Operator *Stat* calculates linkage disequilibrium between the first and second loci. The results of this operator are stored in a local variable space of the Population. The last operator *PyEval* outputs calculated linkage disequilibrium values with a trailing new line. The result represents the decay of linkage disequilibrium of this population at 10 generation intervals. The return value of the `evolve` function, which is the number of evolved generations, is also printed.

2.3 Features

simuPOP offers a long list of features, many of which are unique among all forward-time population genetics simulation programs. The most distinguishing features include:

1. simuPOP provides three types of modules that use 1, 8 or 32/64 bits to store an allele. The binary module (1 bit) is suitable for simulating a large number of SNP markers, and the long module (32 or 64 bits depending on platform) is suitable for simulating some population genetics models such as the infinite allele mutation model.
2. [NEW in simuPOP 1.0.7] simuPOP provides modules to store a large number of rare variants in a compressed manner (the mutant module), and to store origin of each allele so that it is easy to track allelic lineage during evolution.
3. The core of simuPOP is implemented in C++ which is heavily optimized for large-scale simulations. simuPOP can be executed in multiple threads with boosted performance on modern multi-core CPUs.
4. In addition to autosomes and sex chromosomes, simuPOP supports arbitrary types of chromosomes through customizable genotype transmitters. Random maternal transmission of mitochondrial DNAs is supported as a special case of this feature.
5. An arbitrary number of float numbers, called **information fields**, can be attached to individuals of a population. For example, information field `father_idx` and `mother_idx` can be used to track an individual's parents, and `pack_year` can be used to simulate an environmental factor associated with smoking.
6. simuPOP does not impose a limit on the number of homologous sets of chromosomes, the size of the genome or populations. The size of your simulation is only limited by the physical memory of your computer.

7. During an evolutionary process, a population can hold more than one most- recent generation. Pedigrees can be sampled from such multi-generation populations.
8. An operator can be native (implemented in C++) or hybrid (Python-assisted). A hybrid operator calls a user-provided Python function to implement arbitrary genetic effects. For example, a hybrid mutator passes to-be-mutated alleles to a function and mutates these alleles according to the returned values.
9. simuPOP provides more than 60 operators that cover all important aspects of genetic studies. These include mutation (*e.g.* *k*-allele, stepwise, generalized stepwise and context-sensitive mutation models), migration (arbitrary, can create new subpopulation), recombination and gene conversion (uniform or nonuniform), selection (single-locus, additive, multiplicative or hybrid multi- locus models, support selection of both parents and offspring), penetrance (single, multi-locus or hybrid), ascertainment (casecontrol, affected sibpairs, random, nuclear and large Pedigrees), statistics calculation (including but not limited to allele, genotype, haplotype, heterozygote number and frequency; linkage disequilibrium measures, Hardy-Weinberg test), pedigree tracing, visualization (using R or other Python modules) and load/save in simuPOP's native format and many external formats such as Linkage.
10. Mating schemes can work on virtual subpopulations of a subpopulation. For example, positive assortative mating can be implemented by mating individuals with similar properties such as ancestry and overlapping generations could be simulated by copying individuals across generations. The number of offspring per mating event can be fixed or can follow a statistical distribution.

A number of forward-time simulation programs are available. If we exclude early forward-time simulation applications developed primarily for teaching purposes, notable forward-time simulation programs include *easyPOP*, *FPG*, *Nemo* and *quantiNemo*, *genoSIM* and *genomeSIMLA*, *FreGene*, *GenomePop*, *ForwSim*, and *ForSim*. These programs are designed with specific applications and specific evolutionary scenarios in mind, and excel in what they are designed for. For some applications, these programs may be easier to use than simuPOP. For example, using a special look-ahead algorithm, *ForwSim* is among the fastest programs to simulate a standard Wright-Fisher process, and should be used if such a simulation is needed. However, these programs are not flexible enough to be applied to problems outside of their designed application area. For example, none of these programs can be used to study the evolution of a disease predisposing mutant, a process that is of great importance in statistical genetics and genetic epidemiology. Compared to such programs, simuPOP has the following advantages:

- The scripting interface gives simuPOP the flexibility to create arbitrarily complex evolutionary scenarios. For example, it is easy to use simuPOP to explicitly introduce a disease predisposing mutant to an evolving population, trace the allele frequency of them, and restart the simulation if they got lost due to genetic drift.
- The Python interface allows users to define customized genetic effects in Python. In contrast, other programs either do not allow customized effects or force users to modify code at a lower (*e.g.* C++) level.
- simuPOP is the only application that embodies the concept of virtual subpopulation that allows evolutions at a finer scale. This is required for realistic simulations of complex evolutionary scenarios.
- simuPOP allows users to examine an evolutionary process very closely because all simuPOP objects are Python objects that can be assessed using their member functions. For example, users can keep track of genotype at particular loci during evolution. In contrast, other programs work more or less like a black box where only limited types of statistics can be outputted.

2.4 License, Distribution and Installation

simuPOP is distributed under a GPL license and is hosted at <http://simupop.sourceforge.net>, the world's largest development and download repository of Open Source code and applications. simuPOP is available on any platform where Python is available, and is currently tested under both 32 and 64 bit versions of Windows (Windows 2000 and later), Linux (Redhat and Ubuntu), MacOS X and Sun Solaris systems. Different C++ compilers such as Microsoft Visual C++, gcc and Intel icc are supported under different operating systems. Standard installation packages are provided for Windows, Linux, and MacOS X systems.

If a binary distribution is unavailable for a specific platform, it is usually easy to compile simuPOP from source, following the standard `python setup.py install` procedure. Please refer to the installation section of the simupop website for instructions for specific platforms and compilers.

simuPOP is available for Python 2.4 and later, including the new Python 3.x releases. Although Python 3 is incompatible with Python 2 in many ways, examples in this guide are written in a style that is compatible with both versions of Python. Some non-classic usages include the use of `a//b` instead of `a/b` for floored division and `list(range(3))` instead of `range(3)` for sequence `[0, 1, 2]`. In particular, we use

```
print("Population size is %d" % size)
```

instead of

```
print "Population size is %d" % size
```

to output strings because the former is valid in Python 2.x (print a tuple with one element) and will generate the same output in Python 3.x. Of course, users of simuPOP can choose to use other styles.

Thanks to the ‘glue language’ nature of Python, it is easy to inter-operate with other applications within a simuPOP script. For example, users can call any R function from Python/simuPOP for the purposes of visualization and statistical analysis, using R and a Python module `RPy`. Because simuPOP utility modules such as `simuPOP.plotter` and `simuPOP.sampling` makes use of R and `rpy` (not `rpy2`) to plot figures, **it is highly recommended that you install R and RPy with simuPOP**. In addition, although simuPOP uses the standard Tkinter GUI toolkit when a graphical user interface is needed, it can make use of a wxPython toolkit if it is available.

2.5 How to read this user’s guide

This user’s guide describes all simuPOP features using a lot of examples. The first few chapters describes all classes in the simuPOP core. Chapter *cha_simuPOP_Operators* describes almost all simuPOP operators, divided largely by genetic models. Features listed in these two chapters are generally implemented at the C++ level and are provided through the `simuPOP` module. Chapter *cha_Utility_Modules* describes features that are provided by various simuPOP utility modules. These modules provide extensions to the simuPOP core that improves the usability and userfriendliness of simuPOP. The next chapter (Chapter *cha_A_real_example*) demonstrates how to write a script to solve a real-world simulation problem. Because some sections describe advanced features that are only used in the construction of highly complex simulations, or implementation details that concern only advanced users, new simuPOP users can safely skip these sections. **Sections that describe advanced topics are marked by one or two asterisks (*) after the section titles.**

simuPOP is a comprehensive forward-time population genetics simulation environment with many unique features. If you are new to simuPOP, you can go through this guide quickly and understand what simuPOP is and what features it provides. Then, you can read Chapter *cha_A_real_example* and learn how to apply simuPOP in real-world problems. After you play with simuPOP for a while and start to write simple scripts, you can study relevant sections in details. The *simuPOP reference manual* will become more and more useful when the complexity of your scripts grows.

Before we dive into the details of simuPOP, it is helpful to know a few name conventions that simuPOP tries to follow. Generally speaking,

- All class names use the CapWords convention (e.g. `Population()`, `InitSex()`).
- All standalone functions (e.g. `loadPopulation()` and `initSex`), member functions (e.g. `Population.mergeSubPops()`) and parameter names use the mixedCases style.
- Constants are written in all capital characters with underscores separating words (e.g. `CHROMOSOME_X`, `UNIFORM_DISTRIBUTION`). Their names instead of their actual values should be used because those values can change without notice.
- simuPOP uses the abbreviated form of the following words in function and parameter names:

pop (population), pops (populations), pos (position), info (information), migr (migration), subPop (subpopulation and virtual subpopulation), subPops (subpopulations and virtual subpopulations), rep (replicates), gen (generation), ops (operators), expr (expression), stmts (statements).

- simuPOP uses both singular and plural forms of parameters, according to the following rules:
- **If a parameter only accept a single input, singular names such as `field`, `locus`, `value`, and `name` are used.**
- **If a parameter accepts a list of values, plural names such as `fields`, `loci`, `values` and `names` are used. Such parameters usually accept single inputs.** For example, `loci=1` can be used as a shortcut for `loci=[1]` and `infoFields='x'` can be used as a shortcut for `infoFields=['x']`.

The same rules also hold for function names. For example, `Population.addInfoFields()` accept a list of information fields but `pop.addInfoFields('field')` is also acceptable.

2.6 Other help sources

If you are new to Python, it is recommended that you borrow a Python book, or at least go through the following online Python tutorials:

1. The Python tutorial (<http://docs.python.org/tut/tut.html>)
2. Other online tutorials listed at <http://www.python.org/doc/>

If you are new to simuPOP, please read this guide before you dive into *the simuPOP reference manual*, which describes all the details of simuPOP but does not show you how to use them. Both documents are available online at <http://simupop.sourceforge.net> in both searchable HTML format and PDF format.

A *simuPOP online cookbook* (<http://simupop.sourceforge.net/cookbook>) is a wiki-based website where you can browse and download examples, functions and scripts for various simulation scenarios, and upload your own code snippets for the benefit of all simuPOP users. Please consider contributing to this cookbook if you have written some scripts that might be useful to others.

If you cannot find the answer you need, or if you believe that you have encountered a bug, or if you would like to request a feature, please subscribe to the simuPOP mailinglist (simupop-list@lists.sourceforge.net) and send your questions there.

Loading and running simuPOP

3.1 Pythonic issues

3.1.1 `from simuPOP import *` v.s. `import simuPOP`

Generally speaking, it is recommended to use `import simuPOP` rather than `from simuPOP import *` to import a simuPOP module. That is to say, instead of using

```
from simuPOP import *
pop = Population(size=100, loci=[5])
simu = Simulator(pop, RandomMating())
```

it is recommended that you use simuPOP like

```
import simuPOP
pop = simuPOP.Population(size=100, loci=[5])
simu = simuPOP.Simulator(pop, simuPOP.RandomMating())
```

The major problem with `from simuPOP import *` is that it imports all simuPOP symbols to the global namespace and increases the likelihood of name clashes. For example, if you import a module `myModule` after `simuPOP`, which happens to have a variable named `MALE`, the following code might lead to a `TypeError` indicating your input for parameter `sex` is wrong.

```
from simuPOP import *
from myModule import *
pop = Population(size=100, loci=[5])
initSex(pop, sex=[MALE, FEMALE])
```

It can be even worse if the definition of `MALE` is changed to a different value of the same type (e.g. to `FEMALE`) and your simulation might produce erroranous result without a hint.

For the sake of brevity, all examples in this user's guide use `import simuPOP as sim` as an alternative form of the `import simuPOP` style. This saves some keystrokes by referring simuPOP functions as `sim.Population()`

instead of `simuPOP.Population()`. Note that `simuPOP` has a number of submodules, which are not imported by default. The recommended syntax to load these modules is:

```
# import and use submodule simuPOP.utils
from simuPOP import utils
utils.simulateBackwardTrajectory(N=1000, endGen=100, endFreq=0.1)
```

3.1.2 References and the `clone()` member function

Assignment in Python only creates a new reference to an existing object. For example,

```
pop = Population()
pop1 = pop
```

creates a reference `pop1` to population `pop`. Modifying `pop1` will modify `pop` as well and the removal of `pop` will invalidate `pop1`. For example, a reference to the first `Population` in a simulator is returned from function `func()` in Example [lst_Reference_to_Population](#). The subsequent use of this `pop` object may crash `simuPOP` because the simulator `simu` is destroyed, along with all its internal populations, after `func()` is finished, leaving `pop` referring to an invalid object.

Example: *Reference to a population in a simulator*

```
def func():
    simu = Simulator(Population(10), RandomMating(), rep=5)
    # return a reference to the first Population in the simulator
    return simu.population(0)

pop = func()
# simuPOP will crash because pop refers to an invalid Population.
pop.popSize()
```

If you would like to have an independent copy of a population, you can use the `clone()` member function. Example [lst_Reference_to_Population](#) would behave properly if the `return` statement is replaced by

```
return simu.population(0).clone()
```

although in this specific case, extracting the first population from the simulator using the `extract` function

```
return simu.extract(0)
```

would be more efficient.

The `clone()` function exists for all `simuPOP` classes (objects) such as *simulator*, *mating schemes* and *operators*. `simuPOP` also supports the standard Python shallow and deep copy operations so you can also make a cloned copy of `pop` using the `deepcopy` function defined in the Python `copy` module

```
import copy
pop1 = copy.deepcopy(pop)
```

3.1.3 Zero-based indexes, absolute and relative indexes

All arrays in `simuPOP` start at index 0. This conforms to Python and C++ indexes. To avoid confusion, I will refer the first locus as locus zero, the second locus as locus one; the first individual in a population as Individual zero, and so on.

Another two important concepts are the *absolute index* and *relative index* of a locus. The former index ignores chromosome structure. For example, if there are 5 and 7 loci on the first two chromosomes, the absolute indexes of the two chromosomes are (0, 1, 2, 3, 4), (5, 6, 7, 8, 9, 10, 11) and the relative indexes are (0, 1, 2, 3, 4), (0, 1, 2, 3, 4, 5, 6). Absolute indexes are more frequently used because they avoid the trouble of having to use two numbers (chrom, index) to refer to a locus. Two functions `chromLocusPair(idx)` and `absLocusIndex(chrom, index)` are provided to convert between these two kinds of indexes. An individual can also be referred by its *absolute index* and *relative index* where *relative index* is the index in its subpopulation. Related member functions are `subPopIndPair(idx)` and `absIndIndex(idx, subPop)`. Example [absIndex](#) demonstrates the use of these functions.

Example: *Conversion between absolute and relative indexes*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[10, 20], loci=[5, 7])
>>> print(pop.chromLocusPair(7))
(1, 2)
>>> print(pop.absLocusIndex(1, 1))
6
>>> print(pop.absIndIndex(2, 1))
12
>>> print(pop.subPopIndPair(25))
(1, 15)

now exiting runScriptInteractively...
```

[Download absIndex.py](#)

3.1.4 Ranges and iterators

Ranges in simuPOP also conform to Python ranges. That is to say, a range has the form of `[a, b)` where `a` belongs to the range, and `b` does not. For example, `pop.chromBegin(1)` refers to the index of the first locus on chromosome 1 (actually exists), and `pop.chromEnd(1)` refers to the index of the last locus on chromosome 1 **plus 1**, which might or might not be a valid index.

A number of simuPOP functions return Python iterators that can be used to iterate through an internal array of objects. For example, `Population.Individuals([subPop])` returns an iterator iterates through all individuals, or all individuals in a (virtual) subpopulation. `Simulator.populations()` can be used to iterate through all populations in a simulator. Example [iterator](#) demonstrates the use of ranges and iterators in simuPOP.

Example: *Ranges and iterators*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2, loci=[5, 6])
>>> sim.initGenotype(pop, freq=[0.2, 0.3, 0.5])
>>> for ind in pop.individuals():
...     for loc in range(pop.chromBegin(1), pop.chromEnd(1)):
...         print(ind.allele(loc))
...
0
2
2
1
1
1
1
1
2
```

(continues on next page)

(continued from previous page)

```

2
2
1
2

now exiting runScriptInteractively...
```

[Download iterator.py](#)

3.1.5 Empty, ALL_AVAIL and dynamic values for parameters loci, reps, ancGen and subPops

Parameters `loci`, `reps` and `subPops` are widely used in `simuPOP` to specify which loci, replicates, ancestral generations, or (virtual) subpopulations a function or operator is applied to. These parameter accepts a list of indexes such as `[1, 2]`, names such as `['a', 'b']`, and take single form inputs (e.g. `loci=1` is equivalent to `loci=[1]`). For example,

- `Recombinator(loci=[])` recombine at no locus, and
- `Recombinator(loci=1)` recombine at locus 1
- `Recombinator(loci=[1, 2, 4])` recombine at loci 1, 2, and 4
- `Recombinator(loci=[('1', 20), ('1', 25)])` recombine at loci with position 20 and 25 on chromosome 1. This usage is only available for parameter `loci`.
- `Recombinator(loci=['a2', 'a4'])` recombine at loci 'a2' and 'a4'.

The last method is easier to understand in some cases. Moreover, when you use loci names instead of indexes in an operator, this operator can be applied to populations with loci at different locations. For example

```
MaSelector(loci='a2', fitness=[1, 1.01, 1.02])
```

will be applied to locus a2 regardless the actual location of this locus in the population to which this operator is applied.

However, in the majority of the cases, these parameters take a default value `ALL_AVAIL` which applies the function or operator to all available loci, replicates or subpopulations. That is to say, `Recombinator()` or `Recombinator(loci=ALL_AVAIL)` will recombine at all applicable loci, which will vary from population to population. Value `UNSPECIFIED` is sometimes used as default parameter of these parameters, indicating that no value has been specified. Similarly, `subPops=[0, 'Male']` can be used to refer a virtual subpopulation with name 'Male', regardless its virtual subpopulation index.

Besides `subPops=ALL_AVAIL`, which means `subPops=[0, 1, 2, 3]` for a population with 4 subpopulations, `ALL_AVAIL` could also be used as `subPops=[(ALL_AVAIL, 1)]` to specify a specific virtual subpopulation for all subpopulations, or `subPops=[(1, ALL_AVAIL)]` or even `subPops=[(ALL_AVAIL, ALL_AVAIL)]` to specify all virtual subpopulations in specified or all subpopulations. This becomes handy when you, for example, would like to list all male individuals in a population, regardless of number of subpopulations.

3.1.6 User-defined functions and class `WithArgs` *

Some `simuPOP` objects call user-defined functions to perform customized operations. For example, a penetrance operator can call a user-defined function with genotype at specified loci and use its return value to determine the affection status of an individual.

simuPOP uses parameter names to determine which information should be passed to such a function. For example, a *PyOperator* will pass a reference to each offspring to a function defined with parameter *off* (e.g. `func(off)`) and references to offspring and his/her parents to a function defined with parameters *off*, *dad*, and *mom* (e.g. `func(off, dad, mom)`). For example, Example *userFunc* defines a function `func(geno, smoking)` using parameters *geno* and *smoking* so operator *PyPenetrance* will pass genotype at specified loci and value at information field *smoking* to this function.

Example: *Use of user-defined Python function in simuPOP*

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(1000, loci=1, infoFields='smoking')
>>> sim.initInfo(pop, lambda:random.randint(0,1), infoFields='smoking')
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>>
>>> # a penetrance function that depends on smoking
>>> def func(geno, smoking):
...     if smoking:
...         return (geno[0]+geno[1])*0.4
...     else:
...         return (geno[0]+geno[1])*0.1
...
>>> sim.pyPenetrance(pop, loci=0, func=func)
>>> sim.stat(pop, numOfAffected=True)
>>> print(pop.dvars().numOfAffected)
352
>>>

now exiting runScriptInteractively...
```

[Download userFunc.py](#)

However, there are circumstances that you do not know the number or names of parameters in advance so it is difficult to define such a function. For example, your function may use an information field with programmed name `'off'+str(numOffspring)` where *numOffspring* is a parameter. In this case, you can create a wrapper function object using *WithArgs*(*func*, *args*) and list passed arguments in *args* (e.g. *WithArgs*(*func*, *args*=['*geno*', '*off*' + *str*(*numOffspring*)]). As long as simuPOP knows which arguments to pass, your function can be defined in any format you want (e.g. use **args* parameters). Example *WithArgs* provides such an example.

Example: *Specify arguments of user-provided function using function WithArgs*

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(1000, loci=1, infoFields=('x', 'y'))
>>> sim.initInfo(pop, lambda:random.randint(0,1), infoFields=('x', 'y'))
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>>
>>> # a penetrance function that depends on unknown information fields
>>> def func(*fields):
...     return 0.4*sum(fields)
...
>>> # function WithArgs tells PyPenetrance that func accepts fields x, y so that
>>> # it will pass values at fields x and y to func.
>>> sim.pyPenetrance(pop, loci=0, func=sim.WithArgs(func, pop.infoFields()))
>>> sim.stat(pop, numOfAffected=True)
>>> print(pop.dvars().numOfAffected)
427
```

(continues on next page)

(continued from previous page)

```
now exiting runScriptInteractively...
```

[Download WithArgs.py](#)

3.1.7 Exception handling *

As shown in Examples *lst_Use_of_standard_module* and *lst_Use_of_optimized_module*, optimized modules raise less exceptions than standard modules. More specifically, the standard modules check for invalid inputs frequently and raise exceptions (e.g. out of bound loci indexes). In contrast, the optimized modules only raise exceptions where proper values could not be pre-determined (e.g. looking for an individual in a population with an ID). **Only exceptions that are raised in both types of modules are documented in the simuPOP reference manual.**

Generally speaking, **you should avoid using exceptions to direct the logic of your script** (e.g. use a `try ... except ...` statement around a function to find a valid input value). Because the optimized modules might not raise these exceptions, such a script may crash or yield invalid results when an optimized module is used. If you have to use such a structure, please check the reference manual and see whether or not an exception will be raised in optimized modules.

3.2 Loading simuPOP modules

3.2.1 Short, long, binary, mutant and lineage modules and their optimized versions

There are ten flavors of the core simuPOP module: short, long, binary, mutant, and lineage allele modules, and their optimized versions.

- The short allele modules use *8 bits* to store each allele which limits the possible allele states to 256. This is enough most of the times so this is the default module of simuPOP.
- If you need to a large number of allele states to simulate, for example the infinite allele model, you should use the long allele version of the modules, which use *32 or 64 bits* for each allele and can have or possible allele states depending on your platform.
- If you would like to simulate a large number of binary (SNP) markers, binary libraries can save you a lot of RAM because they use *1 bit* for each allele.
- If you are simulating long sequence regions with rare variants, you can use the mutant module. This module uses compression technology that ignores wildtype alleles and is not efficient if you need to traverse all alleles frequently. The maximum allele state is 255 for this module. Because this module stores location and value of each allele, it uses at least $64 + 8$ bits for each allele on a 64 bit system. The complexity of the storage also prevents simultaneous write access to genotypes so this module does not benefit much from running in multi-thread mode.
- If you are interested in tracing the lineage of each allele (e.g. the ID of individuals to whom the allele was introduced), you can use the lineage module for which each allele is attached with information about its origin. The maximum allele state is 255 for this module, and the cost of storing each allele is 8 (value) + 32 (lineage) bits.

Despite of differences in internal memory layout, all these modules have the same interface, although some functions behave differently in terms of functionality and performance.

Standard libraries have detailed debug and run-time validation mechanism to make sure a simulation executes correctly. Whenever something unusual is detected, simuPOP would terminate with detailed error messages. The cost of such run-time validation varies from case to case but can be high under some extreme circumstances. Because of this,

optimized versions for all modules are provided. They bypass most parameter checking and run-time validations and will simply crash if things go wrong. It is recommended that you use standard libraries whenever possible and only use the optimized version when performance is needed and you are confident that your simulation is running as expected.

Examples *lst_Use_of_standard_module* and *lst_Use_of_optimized_module* demonstrate the differences between standard and optimized modules, by executing two invalid commands. A standard module checks all input values and raises exceptions when invalid inputs are detected. An interactive Python session would catch these exceptions and print proper error messages. In contrast, an optimized module returns erroneous results and or simply crashes when such inputs are given.

Example: *Use of standard simuPOP modules*

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, loci=2)
>>> pop.locusPos(10)
Traceback (most recent call last):
  File "/var/folders/ys/gnzk0qbx5wbdgm531v82xxljv5yqy8/T/tmp6boewtoh", line 1, in
-><module>
    #begin_file log/standard.py
IndexError: genoStru.h: 557 absolute locus index (10) out of range of 0 ~ 1
>>> pop.individual(20).setAllele(1, 0)
Traceback (most recent call last):
  File "/var/folders/ys/gnzk0qbx5wbdgm531v82xxljv5yqy8/T/tmp6boewtoh", line 1, in
-><module>
    #begin_file log/standard.py
IndexError: population.h: 566 individual index (20) out of range of 0 ~ 9

now exiting runScriptInteractively...
```

Download [standard.py](#)

Example *lst_Use_of_optimized_module* also demonstrates how to use the *setOptions* function in the *simuOpt* module to control the choice of one of the six simuPOP modules. By specifying one of the values *short*, *long* or *binary* for option *alleleType*, and setting *optimized* to *True* or *False*, the right flavor of module will be chosen when simuPOP is loaded. In addition, option *quiet* can be used suppress the banner message when simuPOP is loaded. An alternative method is to set environmental variable *SIMUALLELETYPE* to *short*, *long* or *binary* to use the standard short, long or binary module, and variable *SIMUOPTIMIZED* to use the optimized modules. Command line options *--optimized* can also be used.

Example: *Use of optimized simuPOP modules*

```
% python
>>> from simuOpt import setOptions
>>> setOptions(optimized=True, alleleType='long', quiet=True)
>>> import simuPOP as sim
>>> pop = sim.Population(10, loci=[2])
>>> pop.locusPos(10)
1.2731974748756028e-313
>>> pop.individual(20).setAllele(1, 0)
Segmentation fault
```

3.2.2 Execution in multiple threads

simuPOP is capable of executing in multiple threads but it by default only makes use of one thread. If you have a multi-core CPU, it is often beneficial to set the number of threads to 2 or more to take advantage of this feature. The recommended number of threads is usually the number of cores of your CPU but you might want to set it to a

lower number to leave room for the execution of other applications. The number of threads used in simuPOP can be controlled in the following ways:

- If an environmental variable `OMP_NUM_THREADS` is set to a positive number, simuPOP will be started with specified number of threads.
- Before simuPOP is imported, you can set the number of threads using function `simuOpt.setOptions(numThreads=x)` where `x` can be a positive number (number of threads) or 0, which is interpreted as the number of cores available for your computer.

The number of threads a simuPOP session is used will be displayed in the banner message when simuPOP is imported, and can be retrieved through `moduleInfo['threads']`.

Although simuPOP can usually benefit from the use of multiple cores, certain features of your script might prevent the execution of simuPOP in multiple threads. For example, if your script uses a sex mode of `GLOBAL_SEX_SEQUENCE` to set the sex of offspring according to the global sequence of sexes (e.g. male, male, female), simuPOP will only use on thread to generate offspring because it is not feasible to assign individual sex from a single source of list across multiple threads.

3.2.3 Graphical user interface

A complete graphical user interface (GUI) for users to interactively construct evolutionary processes is still in the planning stage. However, some simuPOP classes and functions can make use of a GUI to improve user interaction. For example, a parameter input dialog can be constructed automatically from a parameter specification list, and be used to accept user input if class `simuOpt.Params` is used to handle parameters. Other examples include a progress bar `simuPOP.utils.ProgressBar` and a dialog used by function `simuPOP.utils.viewVars` to display a large number of variables. The most notable feature of the use of GUI in simuPOP is that **all functionalities can be achieved without a GUI**. For examples, `simuOpt.getParam` will use a terminal to accept user input interactively and `simuPOP.utils.ProgressBar` will turn to a text-based progress bar in the non-GUI mode.

The use of GUI can be controlled either globally or Individually. First, a global GUI parameter could be set by environmental variable `SIMUGUI`, function `simuOpt.setOptions(gui)` or a command line option `--gui` of a simuPOP scripts. Allowed values include

- `True`: This is the system default value. A GUI is used whenever possible. All GUI-capable functions support `wxPython` so a `wxPython` dialog will be used if `wxPython` is available. Otherwise, `tkInter` based dialogs or text- mode will be used.
- `False`: no GUI will be used. All functions will use text-based implementation. Note that `--gui=False` is commonly used to run scripts in batch mode.
- `wxPython`: Force the use of `wxPython` GUI toolkit.
- `Tkinter`: Force the use of `Tkinter` GUI toolkit.

Individual classes and functions that could make use a GUI usually have their own `gui` parameters, which can be set to override global GUI settings. For example, you could force the use of a text-based progress bar by using `ProgressBar(gui=False)`.

3.3 Online help system

Most of the help information contained in this document and *the simuPOP reference manual* is available from command line. For example, after you install and import the simuPOP module, you can use `help(Population.addInfoField)` to view the help information of member function `addInfoField` of class `Population`.

Example: *Getting help using the `texttt{help()}` function*


```
>>> import simuPOP as sim
>>> help(sim.Population.addInfoFields)
Help on built-in function Population_addInfoFields in module simuPOP._simuPOP_std:

Population_addInfoFields(...)
    Usage:

        x.addInfoFields(fields, init=0)

    Details:

        Add a list of information fields fields to a population and
        initialize their values to init. If an information field already
        exists, it will be re-initialized.

now exiting runScriptInteractively...
```

Download help.py

It is important that you understand that

- The constructor of a class is named `__init__` in Python. That is to say, you should use the following command to display the help information of the constructor of class *Population*:

```
>>> help(Population.__init__)
```

- Some classes are derived from other classes and have access to member functions of their base classes. For example, class *Population* and *Individual* are both derived from class *GenoStruTrait*. Therefore, you can use all *GenoStruTrait* member functions from these classes.

In addition, the constructor of a derived class also calls the constructor of its base class so you may have to refer to the base class for some parameter definitions. For example, parameters *begin*, *end*, *step*, *atetc* are shared by all operators, and are explained in details only in class *BaseOperator*.

3.4 Debug-related functions and operators *

Debug information can be useful when something looks suspicious. By turning on certain debug code, simuPOP will print out some internal information before and during evolution. Functions *turnOnDebug*(code) and *turnOffDebug*(code) could be used to turn on and off some debug information.

For example, the following code might crash simuPOP:

```
>>> Population(1, loci=[100]).individual(0).genotype()
```

It is unclear why this simple command causes us trouble, instead of outputting the genotype of the only *Individual* of this population. However, the reason is clear if you turn on debug information:

Example: *Turn on/off debug information*

```
>>> turnOnDebug(DBG_POPULATION)
>>> Population(1, loci=100).individual(0).genotype()
Constructor of population is called
Destructor of population is called
Segmentation fault (core dumped)
```

`Population(1, loci=[100])` creates a temporary object that is destroyed right after the execution of the command. When Python tries to display the genotype, it will refer to an invalid location. The correct method to print the genotype is to create a persistent population object:

```
>>> pop = Population(1, loci=[100])
>>> pop.individual(0).genotype()
```

Another useful debug code is `DBG_WARNING`. When this code is set, it will output warning messages for some common misuse of simuPOP. For example, it will warn you that population object returned by function `Simulator.population()` is a temporary object that will become invalid once a simulator is changed. If you are new to simuPOP, it is recommended that you use

```
import simuOpt
simuOpt.setOptions(optimized=False, debug='DBG_WARNING')
```

when you develop your script.

Besides functions `turnOnDebug(code)` and `turnOffDebug(code)`, you can set environmental variable `SIMUDEBUB=code` where `code` is a comma separated debug codes. A list of valid debug code could be found in function `moduleInfo['debug']`. Note that debug information is only available in standard (non-optimized) modules.

The amount of output can be overwhelming in some cases which makes it necessary to limit the debug information to certain generations, or triggered by certain conditions. In addition, debugging information may interfere with your regular output so you may want to direct such output to another destination, such as a dedicated file.

Example `debug` demonstrates how to turn on debug information conditionally and turn it off afterwards, using operator `PyOperator`. It also demonstrates how to redirect debug output to a file but redefining system standard error output. Note that “is None” is used to make sure the lambda functions return True so that the evolutionary process can continue after the python operator.

Example: Turn on and off debug information during evolution.

```
>>> import simuPOP as sim
>>> # redirect system stderr
>>> import sys
>>> debugOutput = open('debug.txt', 'w')
>>> old_stderr = sys.stderr
>>> sys.stderr = debugOutput
>>> # start simulation
>>> simu = sim.Simulator(sim.Population(100, loci=1), rep=5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.1, 0.9])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.IfElse('alleleNum[0][0] == 0',
...             ifOps=[
...                 # the is None part makes the function return True
...                 sim.PyOperator(lambda : sim.turnOnDebug("DBG_MUTATOR") is None),
...                 sim.PointMutator(loci=0, allele=0, inds=0),
...             ],
...             elseOps=sim.PyOperator(lambda : sim.turnOffDebug("DBG_MUTATOR") is
... ↪None)),
...     ],
```

(continues on next page)

(continued from previous page)

```

...     gen = 100
... )
(100, 100, 100, 100, 100)
>>> # replace standard stderr
>>> sys.stderr = old_stderr
>>> debugOutput.close()
>>> print(''.join(open('debug.txt').readlines()[5:]))
Mutate locus 0 at ploidy 0 to allele 0 at generation 12
Mutate locus 0 at ploidy 0 to allele 0 at generation 13
Mutate locus 0 at ploidy 0 to allele 0 at generation 15
Mutate locus 0 at ploidy 0 to allele 0 at generation 16
Mutate locus 0 at ploidy 0 to allele 0 at generation 21

now exiting runScriptInteractively...
```

[Download debug.py](#)

3.5 Random number generator *

When simuPOP is loaded, it creates a default random number generator (*RNG*) of type `mt19937` for each thread. It uses a random seed for the first RNG and uses seeds derived from the first seed to initialize RNGs for other threads. The seed is drawn from a system random number generator that guarantees random seeds for all instances of simuPOP even if they are initialized at the same time. After simuPOP is loaded, you can reset this system RNG with a different random number generator (c.f. `moduleInfo['availableRNGs']`) or use a specified seed using function `setRNG(name, seed)`.

`getRNG().seed()` returns the seed of the simuPOP random number generator. It can be used to replay your simulation if `getRNG()` is your only source of random number generator. If you also use the Python `random` module, it is a good practise to set its seed using `random.seed(getRNG().seed())`. Example [randomSeed](#) demonstrates how to use these functions to replay an evolutionary process. simuPOP uses a single seed to initialize multiple random number generators used for different threads (seeds for other threads are determined from the first seed) so you only need to save the head seed (`getRNG().seed()`)

Example: *Use saved random seed to replay an evolutionary process*

```

>>> import simuPOP as sim
>>> import random
>>> def simulate():
...     pop = sim.Population(1000, loci=10, infoFields='age')
...     pop.evolve(
...         initOps=[
...             sim.InitSex(),
...             sim.InitGenotype(freq=[0.5, 0.5]),
...             sim.InitInfo(lambda: random.randint(0, 10), infoFields='age')
...         ],
...         matingScheme=sim.RandomMating(),
...         finalOps=sim.Stat(alleleFreq=0),
...         gen=100
...     )
...     return pop.dvars().alleleFreq[0][0]
>>> seed = sim.getRNG().seed()
>>> random.seed(seed)
```

(continues on next page)

(continued from previous page)

```
>>> print('%.4f' % simulate())
0.5780
>>> # will yield different result
>>> print('%.4f' % simulate())
0.6355
>>> sim.setRNG(seed=seed)
>>> random.seed(seed)
>>> # will yield identical result because the same seeds are used
>>> print('%.4f' % simulate())
0.5780

now exiting runScriptInteractively...
```

[Download randomSeed.py](#)

Individuals and Populations

4.1 Genotypic structure

Genotypic structure refers to structural information shared by all individuals in a population, including number of homologous copies of chromosomes (c.f. `ploidy()`, `ploidyName()`), chromosome types and names (c.f. `numChrom()`, `chromType()`, `chromName()`), position and name of each locus (c.f. `numLoci(ch)`, `locusPos(loc)`, `locusName(loc)`), and axillary information attached to each individual (c.f. `infoField(idx)`, `infoFields()`). In addition to property access functions, a number of utility functions are provided to, for example, look up the index of a locus by its name (c.f. `locusByName()`, `chromBegin()`, `chromLocusPair()`).

In `simuPOP`, locus is a (named) position and alleles are just different numbers at that position. **A locus can be a gene, a nucleotide, or even a deletion, depending on how you define alleles and mutations.** For example,

- A codon can be simulated as a locus with 64 allelic states, or three locus each with 4 allelic states. Alleles in the first case would be codons such as AAC and a mutation event would mutate one codon to another (e.g. AAC -> ACC). Alleles in the second case would be A, C, T or G, and a mutation event would mutate one nucleotide to another (e.g. A -> G).
- You can use 0 and 1 (and the binary module of `simuPOP`) to simulate SNP (single-nucleotide polymorphism) markers and ignore the exact meaning of 0 and 1, or use 0, 1, 2, 3 to simulate different nucleotide (A, C, T, or G) in these markers. The mutation model in the second case would be more complex.
- For microsatellite markers, alleles are usually interpreted as the number of tandem repeats. It would be difficult (though doable) to simulate the expansion and contraction of genome caused by the mutation of microsatellite markers.
- The infinite site and infinite allele mutation models could be simulated using either a continuous sequence of nucleotides with a simple 2-allele mutation model, or a locus with a large number of possible allelic states. It is also possible to simulate an empty region (without any locus) with loci introduced by mutation events.
- If you consider deletion as a special allelic state, you can simulate gene deletions without shrinking a chromosome. For example, a deletion mutation event can set the allelic state of one or more loci to 0, which can no longer be mutated.

- Alleles in different individuals could be interpreted differently. For example, if you would like to simulate major chromosomal mutations such as inversion, you could use a super set of markers for different types of chromosomes and use an indicator (information field) to mark the type of chromosome and which markers are valid. Using virtual subpopulations, these individuals could be handled differently during mating.
- In an implementation of an infinite-sites model, **Individual loci are used to store mutation events**. In this example (Example *infiniteSites*), 100 loci are allocated for each individual and they are used to store mutation events (location of a mutation) that happens in a 10Mb region. Whenever a mutation event happens, its location is stored as an allele of an individual. At the end of the evolution, each individual has a list of mutation events which can be readily translated to real alleles. Similar ideas could be used to simulate the accumulation of recombination events.

In summary, the exact meaning of loci and their alleles are user defined. With appropriate mutation model and mating scheme, it is even possible to simulate phenotypic traits using this mechanism, although it is more natural to use information fields for quantitative traits.

A genotypic structure can be retrieved from *Individual* and *Population* objects. **Because a population consists of individuals of the same type, genotypic information can only be changed for all individuals at the population level.** populations in a simulator usually have the same genotypic structure because they are created by as replicates, but their structure may change during evolution. Example *genostructure* demonstrates how to access genotypic structure functions at the population and individual levels. Note that `lociPos` determines the order at which loci are arranged on a chromosome. Loci positions and names will be rearranged if given `lociPos` is unordered.

Example: *Genotypic structure functions*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2, 3], ploidy=2, loci=[5, 10],
...     lociPos=list(range(0, 5)) + list(range(0, 20, 2)), chromNames=['Chr1', 'Chr2
↪'],
...     alleleNames=['A', 'C', 'T', 'G'])
>>> # access genotypic information from the sim.Population
>>> pop.ploidy()
2
>>> pop.ploidyName()
'diploid'
>>> pop.numChrom()
2
>>> pop.locusPos(2)
2.0
>>> pop.alleleName(1)
'C'
>>> # access from an individual
>>> ind = pop.individual(2)
>>> ind.numLoci(1)
10
>>> ind.chromName(0)
'Chr1'
>>> ind.locusName(1)
''
>>> # utility functions
>>> ind.chromBegin(1)
5
>>> ind.chromByName('Chr2')
1
>>> # loci pos can be unordered within each chromosome
>>> pop = sim.Population(loci=[2, 3], lociPos=[3, 1, 1, 3, 2],
...     lociNames=['loc%d' % x for x in range(5)])
>>> pop.lociPos()
```

(continues on next page)

(continued from previous page)

```
(1.0, 3.0, 1.0, 2.0, 3.0)
>>> pop.lociNames()
('loc1', 'loc0', 'loc2', 'loc4', 'loc3')

now exiting runScriptInteractively...
```

[Download genoStru.py](#)

Note: simuPOP does not assume any unit for loci positions. Depending on your application, it can be basepair (bp), kilo-basepair (kb), mega base pair (mb) or even using genetic-map distance such as centiMorgan. It is your responsibility to interpret and use loci positions properly. For example, recombination rate between two adjacent markers can be specified as the product between their physical distance and a recombination intensity. The scale of this intensity will vary by the unit assumed.

Note: Names of loci, alleles and subpopulations are optional. Empty names will be used if they are not specified. Whereas locusName, subPopName and alleleName always return a value (empty string or specified value) for any locus, subpopulation or allele, respectively, lociNames, subPopNames and alleleNames only return specified values, which can be empty lists.

4.1.1 Haploid, diploid and haplodiploid populations

simuPOP is most widely used to study human (diploid) populations. A large number of mating schemes, operators and population statistics are designed around the evolution of such a population. simuPOP also supports haploid and haplodiploid populations although there are fewer choices of mating schemes and operators. simuPOP can also support other types of populations such as triploid and tetraploid populations, but these features are largely untested due to their limited usage. It is expected that supports for these populations would be enhanced over time with additional dedicated operators and functions.

For efficiency considerations, simuPOP saves the same numbers of homologous sets of chromosomes even if some individuals have different numbers of homologous sets in a population. For example, in a haplodiploid population, because male individuals have only one set of chromosomes, their second homologous set of chromosomes are *unused*, which are labeled as '_', as shown in Example [haplodiploid](#).

Example: *An example of haplodiploid population*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2,5], ploidy=sim.HAPLODIPLOID, loci=[3, 5])
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.dump(pop)
Ploidy: 2 (haplodiploid)
Chromosomes:
1: (AUTOSOME, 3 loci)
   (1), (2), (3)
2: (AUTOSOME, 5 loci)
   (1), (2), (3), (4), (5)
population size: 7 (2 subpopulations with 2, 5 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 2 Individuals:
  0: MU 111 00001 | ____
  1: MU 111 01110 | ____
```

(continues on next page)

(continued from previous page)

```
SubPopulation 1 (), 5 Individuals:
  2: MU 111 11110 | ____
  3: MU 101 11111 | ____
  4: MU 110 11111 | ____
  5: MU 101 11101 | ____
  6: MU 110 11001 | ____

now exiting runScriptInteractively...
```

[Download haplodiploid.py](#)

4.1.2 Autosomes, sex chromosomes, mitochondrial, and other types of chromosomes *

The default chromosome type is autosome, which is the *normal* chromosomes in diploid, and in haploid populations. simuPOP supports four other types of chromosomes, namely *chromosome X*, *chromosome Y*, *mitochondrial*, and **customized** chromosome types. Sex chromosomes are only valid in haploid populations where chromosomes X and Y are used to determine the sex of an offspring. Mitochondrial DNAs can exist in haploid or diploid populations, and are inherited maternally. Customized chromosomes rely on user defined functions and operators to be passed from parents to offspring.

Example `subPopName` shows how to specify different chromosome types, and how genotypes of these special chromosomes are arranged.

Example: *Different chromosome types*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=6, ploidy=2, loci=[3, 3, 3, 2, 2, 4, 4],
...     chromTypes=[sim.AUTOSOME]*2 + [sim.CHROMOSOME_X, sim.CHROMOSOME_Y, sim.
↳MITOCHONDRIAL]
...     + [sim.CUSTOMIZED]*2)
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.dump(pop, structure=False) # does not display genotypic structure information
SubPopulation 0 (), 6 Individuals:
  0: MU 111 000 011 ____ 11 1111 1101 | 110 000 ____ 11 ____ 1111 1011
  1: MU 111 111 101 ____ 11 1110 1011 | 111 011 ____ 11 ____ 1110 1011
  2: MU 110 101 011 ____ 11 1011 0011 | 110 100 ____ 11 ____ 1010 1111
  3: MU 010 011 111 ____ 11 1111 1111 | 110 010 ____ 11 ____ 1111 0111
  4: MU 101 000 111 ____ 01 0111 0100 | 110 111 ____ 00 ____ 0111 0001
  5: MU 111 010 111 ____ 10 0111 1011 | 111 111 ____ 11 ____ 0111 1011

now exiting runScriptInteractively...
```

[Download chromType.py](#)

The evolution of sex chromosomes follow the following rules

- There can be only one X chromosome and one Y chromosome. It is not allowed to have only one kind of sex chromosome.
- The Y chromosome of female individuals are ignored. The second homologous copy of the X chromosome and the first copy of the Y chromosome are ignored for male individuals.
- During mating, female parent pass one of her X chromosome to her offspring, male parent pass chromosome X or Y to his offspring. Recombination is allowed for the X chromosomes of females, but not allowed for males.

- The sex of offspring is determined by the types of sex chromosomes he/she inherits, XX for female, and XY for male.

The evolution of mitochondrial DNAs follow the following rules

- There can be only one copy of mitochondrial DNA, exists for both males and females.
- In a non-haploid population where all chromosomes have multiple homologous copies, only the first copy is used for mitochondrial DNA.
- mtDNAs are inherited maternally

Customized chromosomes are used to model more complex types of chromosomes. They rely on customized operators for inheritance. For example, if you would like to model multiple copies of mitochondrial DNAs (cytohets with multiple organellar chromosomes) in a cell, and the process of genetic drift of somatic cytoplasmic segregation of mtDNAs, you can use multiple customized chromosomes to model multiple cytohets (see section [subsec_Pre_defined_genotype_transmitters](#) for an Example). Figure [fig_chromTypes](#) depicts the possible chromosome structure of two diploid parents, and how offspring chromosomes are formed. It uses two customized chromosomes to model multiple copies of mitochondrial chromosomes that are passed randomly from mother to offspring. The second homologous copy of customized chromosomes are unused in this example.

Figure: *Inheritance of different types of chromosomes in a diploid population*

individuals in this population have five chromosomes, one autosome (A), one X chromosome (X), one Y chromosome (Y) and two customized chromosomes (C). The customized chromosomes model multiple copies of mitochondrial chromosomes that are passed randomly from mother to offspring. Y chromosomes for the female parent, the second copy of chromosome X and the first copy of chromosome Y for the male parent, and the second copy of customized chromosomes are unused (gray chromosome regions). A male offspring inherits one copy of autosome from his mother (with recombination), one copy of autosome from his father (with recombination), an X chromosome from his mother (with recombination), a Y chromosome from his father (without recombination), and two copies of the first customized chromosome.

4.1.3 Information fields

Different kinds of simulations require different kinds of individuals. individuals with only genotype information are sufficient to simulate the basic Wright-Fisher model. Sex is needed to simulate such a model in diploid populations with sex. individual fitness may be needed if selection is induced, and age may be needed if the population is age-structured. In addition, different types of quantitative traits or affection status may be needed to study the impact of genotype on Individual phenotype. Because it is infeasible to provide all such information to an individual, simuPOP keeps genotype, sex (MALE or FEMALE) and affection status as *built-in properties* of an individual, and all others as optional *information fields* (float numbers) attached to each individual.

Information fields can be specified when a population is created, or added later using population member functions. They are essential for proper operation of many simuPOP operators. For example, all selection operators require information field `fitness` to store evaluated fitness values for each individual. Operator *Migrator* uses information field `migrate_to` to store the ID of subpopulation an individual will migrate to. An error will be raised if these operators are applied to a population without needed information fields.

Example: *Basic usage of information fields*

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, loci=[20], ancGen=1,
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*20+[1]*20)
...     ],
```

(continues on next page)



Users/bpeng1/simuPOP/simuPOP/doc/figures/chromType.png

(continued from previous page)

```

...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.Recombinator(rates=0.01),
...             sim.ParentsTagger()
...         ]
...     ),
...     gen = 1
... )
1
>>> pop.indInfo('mother_idx') # mother of all offspring
(9.0, 8.0, 8.0, 0.0, 8.0, 9.0, 8.0, 7.0, 7.0, 9.0)
>>> ind = pop.individual(0)
>>> mom = pop.ancestor(ind.mother_idx, 1)
>>> print(ind.genotype(0))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> print(mom.genotype(0))
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> print(mom.genotype(1))
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

now exiting runScriptInteractively...

```

Download infoField.py

Example *basicInfoFields* demonstrates the basic usage of information fields. In this example, a population with two information fields `mother_idx` and `father_idx` are created. Besides the present generation, this population keeps one ancestral generations (`ancGen=1`, see Section *subsec_Ancestral_populations* for details). After initializing each individual with two chromosomes with all zero and all one alleles respectively, the population evolves one generation, subject to recombination at rate 0.01. Parents of each individual are recorded, by operator *ParentsTagger*, to information fields `mother_idx` and `father_idx` of each offspring.

After evolution, the population is extracted from the simulator, and the values of information field `mother_idx` of all individuals are printed. The next several statements get the first Individual from the population, and his mother from the parental generation using the indexes stored in this individual's information fields. Genotypes at the first homologous copy of this individual's chromosome is printed, along with two parental chromosomes.

Information fields can only be added or removed at the population level because all individuals need to have the same set of fields. Values of information fields could be accessed at Individual or population levels, using functions such as *Individual.info*, *Individual.setInfo*, *population.indInfo*, *Population.setIndInfo*. These functions will be introduced in their respective classes.

Note: Information fields can be located both by names and by indexes**, ** the former provides better readability at a slight cost of performance because these names have to be translated into indexes each time. However, use of names are recommended in most cases for readability considerations.

4.2 Individual

individuals are building blocks of a population. An individual object cannot be created independently, but references to individuals can be retrieved using member functions of a population object.

4.2.1 Access individual genotype

From a user's point of view, 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 chromosome, 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. The memory layout of a diploid individual with two chromosomes is illustrated in Figure [fig_genotype_layout](#).

Figure: *Memory layout of individual genotype*

simuPOP provides several functions to read/write individual genotype. For example, `Individual.allele()` and `Individual.setAllele()` can be used to read and write single alleles. You could also access alleles in batch mode using functions `Individual.genotype()` and `Individual.setGenotype()`. It is worth noting that, instead of copying genotypes of an individual to a Python tuple or list, the return value of function `genotype([p, [ch]])` is a special python carray object that reflects the underlying genotypes. This object behaves like a regular Python list except that the underlying genotype will be changed if elements of this object are changed. Only `count(x)` and `index(x, [start, [stop]])` member functions can be used, but all comparison, assignment and slice operations are allowed. If you would like to copy the content of this carray to a Python list, use the `list` function. Example [individualGenotype](#) demonstrates the use of these functions.

Example: *Access individual genotype*

```
>>> import simuPOP as sim
>>> pop = sim.Population([2, 1], loci=[2, 5])
>>> for ind in pop.individuals(1):
...     for marker in range(pop.totNumLoci()):
...         ind.setAllele(marker % 2, marker, 0)
...         ind.setAllele(marker % 2, marker, 1)
...         print('%d %d ' % (ind.allele(marker, 0), ind.allele(marker, 1)))
...
0 0
1 1
0 0
1 1
0 0
1 1
0 0
>>> ind = pop.individual(1)
>>> geno = ind.genotype(1)      # the second homologous copy
>>> geno
[0, 0, 0, 0, 0, 0, 0]
>>> geno[2] = 3
>>> ind.genotype(1)
[0, 0, 3, 0, 0, 0, 0]
>>> geno[2:4] = [3, 4]          # direct modification of the underlying genotype
>>> ind.genotype(1)
[0, 0, 3, 4, 0, 0, 0]
>>> # set genotype (genotype, ploidy, chrom)
>>> ind.setGenotype([2, 1], 1, 1)
>>> geno
[0, 0, 2, 1, 2, 1, 2]
>>> #
>>> geno.count(1)              # count
2
>>> geno.index(2)              # index
2
```

(continues on next page)



Users/bpeng1/simuPOP/simuPOP/doc/figures/genotype.png

(continued from previous page)

```

>>> ind.setAllele(5, 3)      # change underlying genotype using setAllele
>>> print(geno)              # geno is change
[0, 0, 2, 1, 2, 1, 2]
>>> print(geno)              # but not geno
[0, 0, 2, 1, 2, 1, 2]
>>> geno[2:5] = 4            # can use regular Python slice operation
>>> print(ind.genotype())
[0, 0, 0, 5, 0, 0, 0, 0, 0, 4, 4, 4, 1, 2]

now exiting runScriptInteractively...

```

[Download individualGenotype.py](#)

The same object will also be returned by function `Population.genotype()`.

4.2.2 individual sex, affection status and information fields

In addition to structural information shared by all individuals in a population, the individual class provides member functions to get and set *genotype*, *sex*, *affection status* and *information fields* of an individual. Example [individuals](#) demonstrates how to access and modify individual sex, affection status and information fields. Note that **information fields can be accessed as attributes of individuals**. For example, `ind.info('father_idx')` is equivalent to `ind.father_idx` and `ind.setInfo(35, 'age')` is equivalent to `ind.age = 35`.

Example: *Access Individual properties*

```

>>> import simuPOP as sim
>>> pop = sim.Population([5, 4], loci=[2, 5], infoFields='x')
>>> # get an individual
>>> ind = pop.individual(3)
>>> ind.ploidy()              # access to genotypic structure
2
>>> ind.numChrom()
2
>>> ind.affected()
False
>>> ind.setAffected(True)     # access affection sim.status,
>>> ind.sex()                 # sex,
1
>>> ind.setInfo(4, 'x')       # and information fields
>>> ind.x = 5                  # the same as ind.setInfo(4, 'x')
>>> ind.info('x')             # get information field x
5.0
>>> ind.x                     # the same as ind.info('x')
5.0

now exiting runScriptInteractively...

```

[Download individual.py](#)

4.3 Population

The `Population` object is the most important object of `simuPOP`. It consists of one or more generations of individuals, grouped by subpopulations, and a local Python dictionary to hold arbitrary population information. This class

provides a large number of functions to access and modify population structure, individuals and their genotypes and information fields. The following sections explain these features in detail.

4.3.1 Access and change individual genotype

From a user's point of view, genotypes of all individuals in a population are arranged sequentially. Similar to functions `Individual.genotype()` and `Individual.setGenotype()`, genotypes of a population can be accessed in batch using functions `Population.genotype()` and `Population.setGenotype()`. However, because it is error prone to locate an allele of a particular individual in this long array, these functions are usually used to perform population-level genotype operations such as clearing all alleles (e.g. `pop.setGenotype(0)`) or counting the number of a particular allele across all individuals (e.g. `pop.genotype().count(1)`).

Another way to change alleles across the whole population is to recode existing alleles to other numbers. This is sometimes needed if you need to change allele states to conform with a particular mutation model, assumptions of other software applications or genetic samples. For example, if your dataset uses 1, 2, 3, 4 for A, C, T, G alleles, and you would like to use alleles 0, 1, 2 and 3 for A, C, G, T (a convention for simuPOP when nucleotide mutation models are involved), you can use

```
pop.recodeAlleles([0, 0, 1, 3, 2], alleleNames=['A', 'C', 'G', 'T'])
```

to convert and rename the alleles (1 allele to 0, 2 allele to 1, etc). This operation will be applied to all subpopulations for all ancestral generations, but can be restricted to selected loci.

4.3.2 Subpopulations

A simuPOP population consists of one or more subpopulations. **If a population is not structured, it has one subpopulation that is the population itself.** Subpopulations serve as barriers of individuals in the sense that mating only happens between individuals in the same subpopulation. A number of functions are provided to merge, remove, resize subpopulations, and move individuals between subpopulations (migration).

Example `subPopName` demonstrates how to use some of the subpopulation related functions.

Example: *Manipulation of subpopulations*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[3, 4, 5], ploidy=1, loci=1, infoFields='x')
>>> # individual 0, 1, 2, ... will have an allele 0, 1, 2, ...
>>> pop.setGenotype(range(pop.popSize()))
>>> #
>>> pop.subPopSize(1)
4
>>> # merge subpopulations
>>> pop.mergeSubPops([1, 2])
1
>>> # split subpopulations
>>> pop.splitSubPop(1, [2, 7])
(1, 2)
>>> pop.subPopSizes()
(3, 2, 7)
>>> # remove subpopulations
>>> pop.removeSubPops(1)
>>> pop.subPopSizes()
(3, 7)

now exiting runScriptInteractively...
```

Download subPop.py

Some population operations change the IDs of subpopulations. For example, if a population has three subpopulations 0, 1, and 2, and subpopulation 1 is split into two subpouplations, subpopulation 2 will become subpopulation 3. Tracking the ID of a subpopulation can be problematic, especially when conditional or random subpopulation operations are involved. In this case, you can specify names to subpopulations. These names will follow their associated subpopulations during population operations so you can identify the ID of a subpopulation by its name. Note that simuPOP allows duplicate subpopulation names.

Example: *Use of subpopulation names*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[3, 4, 5], subPopNames=['x', 'y', 'z'])
>>> pop.removeSubPops([1])
>>> pop.subPopNames()
('x', 'z')
>>> pop.subPopByName('z')
1
>>> pop.splitSubPop(1, [2, 3])
(1, 2)
>>> pop.subPopNames()
('x', 'z', 'z')
>>> pop.setSubPopName('z-1', 1)
>>> pop.subPopNames()
('x', 'z-1', 'z')
>>> pop.subPopByName('z')
2

now exiting runScriptInteractively...
```

Download subPopName.py

4.3.3 Virtual subpopulations and virtual splitters *

simuPOP subpopulations can be further divided into virtual subpopulations (VSP), which are groups of individuals who share certain properties. For example, all male individuals, all unaffected individuals, all individuals with information field age > 20, all individuals with genotype 0, 0 at a given locus, can form VSPs. VSPs do not have to add up to the whole subpopulation, nor do they have to be non-overlapping. Unlike subpopulations that have strict boundaries, VSPs change easily with the changes of individual properties.

VSPs are defined by virtual splitters. **It is a definition for groups of individuals in each subpopulation.** A splitter defines the same number of VSPs in all subpopulations, although sizes of these VSPs vary across subpopulations due to subpopulation differences. For example, a *SexSplitter*() defines two VSPs, the first with all male individuals and the second with all female individuals, and a *InfoSplitter*(field='x', values=[1, 2, 4]) defines three VSPs whose members have values 1, 2 and 4 at information field x, respectively. This splitter also allows the use of cutoff values and ranges to define VSPs. If different types of VSPs are needed, a combined splitter can be used to combine VSPs defined by several splitters.

A VSP is represented by a [sp, vsp] pair where sp and vsp can be subpopulation indexes or names. Its name and size can be obtained using functions subPopName() and subPopSize(). Example *virtualSplitter* demonstrates how to apply virtual splitters to a population, and how to check VSP names and sizes.

Example: *Define virtual subpopulations in a population*

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[200, 400], loci=[30], infoFields='x')
```

(continues on next page)

(continued from previous page)

```

>>> # assign random information fields
>>> sim.initSex(pop)
>>> sim.initInfo(pop, lambda: random.randint(0, 3), infoFields='x')
>>> # define a virtual splitter by sex
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.numVirtualSubPop()      # Number of defined VSPs
2
>>> pop.subPopName([0, 0])      # Each VSP has a name
'Male'
>>> pop.subPopSize([0, 1])      # Size of VSP 1 in subpopulation 0
109
>>> pop.subPopSize([0, 'Female']) # Refer to vsp by its name
109
>>> # define a virtual splitter by information field 'x'
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='x', values=[0, 1, 2, 3]))
>>> pop.numVirtualSubPop()      # Number of defined VSPs
4
>>> pop.subPopName([0, 0])      # Each VSP has a name
'x = 0'
>>> pop.subPopSize([0, 0])      # Size of VSP 0 in subpopulation 0
46
>>> pop.subPopSize([1, 0])      # Size of VSP 0 in subpopulation 1
92

now exiting runScriptInteractively...

```

Download virtualSplitter.py

VSP provides an easy way to access groups of individuals in a subpopulation and allows finer control of an evolutionary process. For example, mating schemes can be applied to VSPs which makes it possible to apply different mating schemes to, for example, individuals with different ages. By applying migration, mutation etc to VSPs, it is easy to implement advanced features such as sex-biased migrations, different mutation rates for individuals at different stages of a disease. Example *virtualSubPop* demonstrates how to initialize genotype and information fields to individuals in male and female VSPs.

Example: Applications of virtual subpopulations

```

>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(10, loci=[2, 3], infoFields='Sex')
>>> sim.initSex(pop)
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> # initialize male and females with different genotypes.
>>> sim.initGenotype(pop, genotype=[0]*5, subPops=[(0, 0)])
>>> sim.initGenotype(pop, genotype=[1]*5, subPops=[(0, 1)])
>>> # set Sex information field to 0 for all males, and 1 for all females
>>> pop.setIndInfo([sim.MALE], 'Sex', [0, 0])
>>> pop.setIndInfo([sim.FEMALE], 'Sex', [0, 1])
>>> # Print individual genotypes, followed by values at information field Sex
>>> sim.dump(pop, structure=False)
SubPopulation 0 (), 10 Individuals:
  0: FU 11 111 | 11 111 | 2
  1: FU 11 111 | 11 111 | 2
  2: MU 00 000 | 00 000 | 1
  3: MU 00 000 | 00 000 | 1
  4: MU 00 000 | 00 000 | 1
  5: MU 00 000 | 00 000 | 1

```

(continues on next page)

(continued from previous page)

```

6: MU 00 000 | 00 000 | 1
7: FU 11 111 | 11 111 | 2
8: FU 11 111 | 11 111 | 2
9: FU 11 111 | 11 111 | 2

now exiting runScriptInteractively...
```

[Download virtualSubPop.py](#)

4.3.4 Advanced virtual subpopulation splitters **

simuPOP provides a number of virtual splitters that can define VSPs using specified properties. For example, `InfoSplitter(field='a', values=[1, 2, 3])` defines three VSPs whose individuals have values 1, 2, and 3 at information field `a`, respectively; `SexSplitter()` defines two VSPs of male and female individuals, respectively; and `RangeSplitter(ranges=[[0, 2000], [2000, 5000]])` defines two VSPs using two blocks of individuals.

A `CombinedSplitter` can be used if your simulation needs more than one sets of VSPs. For example, you may want to split your subpopulations both by sex and by affection status. In this case, you can define a combined splitter using

```
CombinedSplitter(splitters=[SexSplitter(), AffectionSplitter()])
```

This splitter simply stacks VSPs defined in `AffectionSplitter()` after `SexSplitter()` so that unaffected and affected VSPs are now VSPs 2 and 3 (0 and 1 are used for male and female VSPs).

There are also scenarios when you would like to define finer VSPs with individuals belonging to more than one VSPs. For example, you may want to have a look of frequencies of certain alleles in affected male vs affected females, or count the number of males and females with certain value at an information field. In this case, a `ProductSplitter` can be used to define VSPs using interactions of several VSPs. For example,

```
ProductSplitter(splitters=[SexSplitter(), AffectionSplitter()])
```

defines 4 subpopulations by splitting VSPs defined by `SexSplitter()` with affection status. These four VSPs will then have unaffected male, affected male, unaffected female and affected female individuals, respectively.

If you consider `ProductSplitter` as an intersection splitter that defines new VSPs as intersections of existing VSPs, you may wonder how to define unions of VSPs. For example, you can make a case where you want to consider Individuals with information field `a` < 0 or `a` > 100 together. A regular `InfoSplitter(field='a', cutoff=[0, 100])` cannot do that because it defines three VSPs with , and , respectively. The trick here is to use parameter `vspMap` of a `CombinedSplitter`. If this parameter is defined, multiple VSPs could be groups or reordered to define a new set of VSPs. For example,

```
CombinedSplitter(splitters=[InfoSplitter(field='a', cutoff=[0, 100])], vspMap=[[0, 2], [1, 0], [2, 1]])
```

defines two VSPs using VSPs 0 and 2, and VSP 1 defined by the `InfoSplitter` so that the first VSP contains individuals with or .

Example [advancedVSP](#) demonstrates some advanced usages of virtual splitters.

Example: *Advanced virtual subpopulation usages.*

```

>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[2000, 4000], loci=[30], infoFields='x')
>>> # assign random information fields
>>> sim.initSex(pop)
>>> sim.initInfo(pop, lambda: random.randint(0, 3), infoFields='x')
>>> #
>>> # 1, use a combined splitter
>>> pop.setVirtualSplitter(sim.CombinedSplitter(splitters = [
...     sim.SexSplitter(),
...     sim.InfoSplitter(field='x', values=[0, 1, 2, 3])
... ]))
>>> pop.numVirtualSubPop()      # Number of defined VSPs
6
>>> pop.subPopName([0, 0])      # Each VSP has a name
'Male'
>>> pop.subPopSize([0, 0])      # sim.MALE
1011
>>> pop.subPopSize([1, 4])      # individuals in sp 1 with value 2 at field x
1048
>>> #
>>> # use a product splitter that defines additional VSPs by sex and info
>>> pop.setVirtualSplitter(sim.ProductSplitter(splitters = [
...     sim.SexSplitter(names=['M', 'F']), # give a new set of names
...     sim.InfoSplitter(field='x', values=[0, 1, 2, 3])
... ]))
>>> pop.numVirtualSubPop()      # Number of defined VSPs
8
>>> pop.subPopName([0, 0])      # Each VSP has a name
'M, x = 0'
>>> pop.subPopSize([0, 0])      # sim.MALE with value 1 in sp 0
240
>>> pop.subPopSize([1, 5])      # sim.FEMALE with value 1 in sp 1
453
>>> #
>>> # use a combined splitter to join VSPs defined by a
>>> # product splitter
>>> pop.setVirtualSplitter(sim.CombinedSplitter([
...     sim.ProductSplitter([
...         sim.SexSplitter(),
...         sim.InfoSplitter(field='x', values=[0, 1, 2, 3])
... ])],
...     vspMap = [[0,1,2], [4,5,6], [7]],
...     names = ['Male x<=3', 'Female x<=3', 'Female x=4'])
>>> pop.numVirtualSubPop()      # Number of defined VSPs
3
>>> pop.subPopName([0, 0])      # Each VSP has a name
'Male x<=3'
>>> pop.subPopSize([0, 0])      # sim.MALE with value 0, 1, 2 at field x
770
>>> pop.subPopSize([1, 1])      # sim.FEMALE with value 0, 1 or 2 at field x
1493

now exiting runScriptInteractively...

```

[Download advancedVSP.py](#)

4.3.5 Access individuals and their properties

There are many ways to access individuals of a population. For example, function `Population.Individual(idx)` returns a reference to the `idx`-th individual in a population. An optional parameter `subPop` can be specified to return the `idx`-th individual in the `subPop`-th subpopulation.

If you would like to access a group of individuals, either from a whole population, a subpopulation, or from a virtual subpopulation, `Population.individuals([subPop])` is easier to use. This function returns a Python iterator that can be used to iterate through individuals. An advantage of this function is that `subPop` can be a virtual subpopulation which makes it easy to iterate through Individuals with certain properties (such as all male Individuals). If you would like to iterate through multiple virtual subpopulations in one or more ancestral generations, you can use another function `Population.allIndividuals(subPops, ancGens)`.

If more than one generations are stored in a population, function `ancestor(idx, [subPop], gen)` can be used to access Individual from an ancestral generation (see Section [subsec_Ancestral_populations](#) for details). Because there is no group access function for ancestors, it may be more convenient to use `useAncestralGen` to make an *ancestral* generation the *current* generation, and use `Population.Individuals`. Note that `ancestor()` function can always access individuals at a certain generation, regardless which generation the current generation is. Example [batchAccess](#) demonstrates how to use all these Individual-access functions.

If an unique ID is assigned to all individuals in a population, you can look up individuals from their IDs using function `Population.indByID()`. The information field to save individual ID is usually `ind_id` and you can use operator `IDTagger` and its function form `tagID` to set this field. Note that this function can be used to look up individuals in the present and all ancestral generations, although a parameter (`ancGen`) can be used to limit search to a specific generation if you know in advance which generation the individual locates.

Example: *Access individuals of a population*

```
>>> import simuPOP as sim
>>> # create a sim.population with two generations. The current generation has values
>>> # 0-9 at information field x, the parental generation has values 10-19.
>>> pop = sim.Population(size=[5, 5], loci=[2, 3], infoFields='x', ancGen=1)
>>> pop.setIndInfo(range(10, 20), 'x')
>>> pop1 = pop.clone()
>>> pop1.setIndInfo(range(10), 'x')
>>> pop.push(pop1)
>>> #
>>> ind = pop.individual(5)          # using absolute index
>>> ind.x
5.0
>>> ind.x          # the same as ind.x
5.0
>>> # use a for loop, and relative index
>>> for idx in range(pop.subPopSize(1)):
...     print(pop.individual(idx, 1).x)
...
5.0
6.0
7.0
8.0
9.0
>>> # It is usually easier to use an iterator
>>> for ind in pop.individuals(1):
...     print(ind.x)
...
5.0
6.0
7.0
```

(continues on next page)

(continued from previous page)

```

8.0
9.0
>>> # Access individuals in VSPs
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=[3, 7, 17], field='x'))
>>> for ind in pop.individuals([1, 1]):
...     print(ind.x)
...
5.0
6.0
>>> # Access all individuals in all ancestral generations
>>> print([ind.x for ind in pop.allIndividuals()])
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0,
↪ 16.0, 17.0, 18.0, 19.0]
>>> # or only specified subpopulations or ancestral generations
>>> print([ind.x for ind in pop.allIndividuals(subPops=[(0,2), (1,3)], ancGens=1)])
[10.0, 11.0, 12.0, 13.0, 14.0, 17.0, 18.0, 19.0]
>>>
>>> # Access individuals in ancestral generations
>>> pop.ancestor(5, 1).x      # absolute index
15.0
>>> pop.ancestor(0, 1, 1).x   # relative index
15.0
>>> # Or make ancestral generation the current generation and use 'individual'
>>> pop.useAncestralGen(1)
>>> pop.individual(5).x       # absolute index
15.0
>>> pop.individual(0, 1).x    # relative index
15.0
>>> # 'ancestor' can still access the 'present' (generation 0) generation
>>> pop.ancestor(5, 0).x
5.0
>>> # access individual by ID
>>> pop.addInfoFields('ind_id')
>>> sim.tagID(pop)
>>> [int(ind.ind_id) for ind in pop.individuals()]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> # access individual by ID. Note that individual 12 is in the parental generation
>>> pop.indByID(12).x
1.0

now exiting runScriptInteractively...

```

Download `accessIndividual.py`

Although it is easy to access individuals in a population, it is often more efficient to access genotypes and information fields in batch mode. For example, functions `genotype()` and `setGenotype()` can read/write genotype of all individuals in a population or (virtual) subpopulation, functions `indInfo()` and `setIndInfo()` can read/write certain information fields in a population or (virtual) subpopulation. The write functions work in a circular manner in the sense that provided values are reused if they are not enough to fill all genotypes or information fields. Example [batchAccess](#) demonstrates the use of such functions.

Example: Access Individual properties in batch mode

```

>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[4, 6], loci=2, infoFields='x')
>>> pop.setIndInfo([random.randint(0, 10) for x in range(10)], 'x')

```

(continues on next page)

(continued from previous page)

```

>>> pop.indInfo('x')
(7.0, 5.0, 8.0, 10.0, 7.0, 0.0, 8.0, 4.0, 4.0, 10.0)
>>> pop.setGenotype([0, 1, 2, 3], 0)
>>> pop.genotype(0)
[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=[3], field='x'))
>>> pop.setGenotype([0])      # clear all values
>>> pop.setGenotype([5, 6, 7], [1, 1])
>>> pop.indInfo('x', 1)
(7.0, 0.0, 8.0, 4.0, 4.0, 10.0)
>>> pop.genotype(1)
[5, 6, 7, 5, 0, 0, 0, 0, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6]

now exiting runScriptInteractively...

```

[Download batchAccess.py](#)

4.3.6 Attach arbitrary auxiliary information using information fields

Information fields are usually set during population creation, using the `infoFields` parameter of the population constructor. It can also be set or added using functions `setInfoFields`, `addInfoField` and `addInfoFields`. Example *popInfo* demonstrates how to read and write information fields from an individual, or from a population in batch mode. Note that functions *Population.indInfo* and *Population.setIndInfo* can be applied to (virtual) subpopulation using an optional parameter `subPop`.

Example: *Add and use of information fields in a population*

```

>>> import simuPOP as sim
>>> pop = sim.Population(10)
>>> pop.setInfoFields(['a', 'b'])
>>> pop.addInfoFields('c')
>>> pop.addInfoFields(['d', 'e'])
>>> pop.infoFields()
('a', 'b', 'c', 'd', 'e')
>>> #
>>> # information fields can be accessed in batch mode
>>> pop.setIndInfo([1], 'c')
>>> # as well as individually.
>>> for ind in pop.individuals():
...     ind.e = ind.c + 1
...
>>> print(pop.indInfo('e'))
(2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)

now exiting runScriptInteractively...

```

[Download popInfo.py](#)

4.3.7 Keep track of ancestral generations

A simuPOP population usually holds individuals in one generation. During evolution, an offspring generation will replace the parental generation and become the present generation (population), after it is populated from a parental population. The parental generation is discarded.

This is usually enough when only the present generation is of interest. However, parental generations can provide useful information on how genotype and other information are passed from parental to offspring generations. simuPOP provides a mechanism to store and access arbitrary number of ancestral generations in a population object. Applications of this feature include pedigree tracking, reconstruction, and pedigree ascertainment.

A parameter `ancGen` is used to specify how many generations a population object *can* store (which is usually called the *ancestral depth* of a population). This parameter is default to 0, meaning keeping no ancestral population. You can specify a positive number `n` to store `n` most recent generations; or -1 to store all generations. Of course, storing all generations during an evolutionary process is likely to exhaust the RAM of your computer quickly.

Several member functions can be used to manipulate ancestral generations:

- `ancestralGens()` returns the number of ancestral generations stored in a population.
- `setAncestralDepth(depth)` resets the number of generations a population can store.
- `push(pop)` will push population `pop` into the current population. `pop` will become the current generation, and the current generation will either be removed (if `ancGen == 0`), or become the parental generation of `pop`. The greatest ancestral generation may be removed. This function is rarely used because populations with ancestral generations are usually created during an evolutionary process.
- `useAncestralGen(idx)` set the present generation to `idx` generation. `idx = 1` for the parental generation, 2 for grand-parental, ..., and 0 for the present generation. This is useful because most population functions act on the *present* generation. You should always call `setAncestralPop(0)` after you examined the ancestral generations.

If a population has several ancestral generations, they are referred by their indexes 0 (the latest generation), 1 (parental generation), ... and (top-most ancestral generation) where equals to `ancestralGens()`. In many cases, you can retrieve the properties of ancestral generations directly, using functions such as

- `popSize(ancGen=-1)`, `subPopSizes(ancGen=-1)`, `subPopSize(subPop, ancGen=-1)`: population and subpopulation sizes of ancestral generation `ancGen`.
- `ancestor(index, ancGen)`: Get a reference to the `index` individual of ancestral generation `ancGen`.

However, most population member functions work at the current generation so you will need to switch to an ancestral generation using function `useAncestralGen()` if you would like to manipulate an ancestral generation. For example, you can remove the second subpopulation of the parental generation using functions:

```
pop.useAncestralGen(1)
pop.removeSubPops(1)
```

A typical use of ancestral generations is demonstrated in example [extract](#). In this example, a population is created and is initialized with allele frequency 0.5. Its ancestral depth is set to 2 at the beginning of generation 18 so that it can hold parental generations at generation 18 and 19. The allele frequency at each generation is calculated and displayed, both during evolution using a *Stat* operator, and after evolution using the function form this operator. Note that setting the ancestral depth at the end of an evolutionary process is a common practice because we are usually only interested in the last few generations.

Example: Ancestral populations

```
>>> import simuPOP as sim
>>> pop = sim.Population(500, loci=1, ancGen=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme = sim.RandomMating(),
...     postOps=[
```

(continues on next page)

(continued from previous page)

```

...     sim.Stat(alleleFreq=0, begin=-3),
...     sim.PyEval(r"%.3f\n" % alleleFreq[0][0], begin=-3)
...     ],
...     gen = 20
... )
0.495
0.510
0.506
20
>>> # information
>>> pop.ancestralGens()
2
>>> pop.popSize(ancGen=1)
500
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> # number of males in the current and parental generation
>>> pop.subPopSize((0,0), pop.subPopSize((0,0), ancGen=1)
(254, 249)
>>> # start from current generation
>>> for i in range(pop.ancestralGens(), -1, -1):
...     pop.useAncestralGen(i)
...     sim.stat(pop, alleleFreq=0)
...     print('%d    %.3f' % (i, pop.dvars().alleleFreq[0][0]))
...
2    0.495
1    0.510
0    0.506
>>> # restore to the current generation
>>> pop.useAncestralGen(0)

now exiting runScriptInteractively...
```

[Download ancestralPop.py](#)

4.3.8 Change genotypic structure of a population

Several functions are provided to remove, add empty loci or chromosomes, and to merge loci or chromosomes from another population. They can be used to trim unneeded loci, expand existing population or merge two populations. Example *extract* demonstrates how to use these populations. Note that function *Population.addLociFrom* by default merges chromosomes one by one according to chromosome index. If *byName* is set to True, it will try to match chromosomes by name and merge them. This example also demonstrates the use of *DBG_WARNING* flag, which will trigger a warning message when chromosomes with different names are merged.

Example: *Add and remove loci and chromosomes*

```

>>> import simuOpt
>>> simuOpt.setOptions(debug='DBG_WARNING')
>>> import simuPOP as sim
Turn on debug 'DBG_WARNING'
>>> pop = sim.Population(10, loci=3, chromNames=['chr1'])
>>> # 1 1 1,
>>> pop.setGenotype([1])
>>> # 1 1 1, 0 0 0
>>> pop.addChrom(lociPos=[0.5, 1, 2], lociNames=['rs1', 'rs2', 'rs3'],
...     chromName='chr2')
```

(continues on next page)

(continued from previous page)

```

>>> pop1 = sim.Population(10, loci=3, chromNames=['chr3'],
...     lociNames=['rs4', 'rs5', 'rs6'])
>>> # 2 2 2,
>>> pop1.setGenotype([2])
>>> # 1 1 1, 0 0 0, 2 2 2
>>> pop.addChromFrom(pop1)
>>> # 1 1 1, 0 0 0, 2 0 2 2 0
>>> pop.addLoci(chrom=[2, 2], pos=[1.5, 3.5], lociNames=['rs7', 'rs8'])
(7, 10)
>>> # 1 1 1, 0 0 0, 2 0 2 0
>>> pop.removeLoci(8)
>>> # loci names can also be used.
>>> pop.removeLoci(['rs1', 'rs7'])
>>> sim.dump(pop)
Ploidy: 2 (diploid)
Chromosomes:
1: chr1 (AUTOSOME, 3 loci)
   (1), (2), (3)
2: chr2 (AUTOSOME, 2 loci)
   rs2 (1), rs3 (2)
3: chr3 (AUTOSOME, 3 loci)
   rs4 (1), rs6 (3), rs8 (3.5)
population size: 10 (1 subpopulations with 10 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 10 Individuals:
0: MU 111 00 220 | 111 00 220
1: MU 111 00 220 | 111 00 220
2: MU 111 00 220 | 111 00 220
3: MU 111 00 220 | 111 00 220
4: MU 111 00 220 | 111 00 220
5: MU 111 00 220 | 111 00 220
6: MU 111 00 220 | 111 00 220
7: MU 111 00 220 | 111 00 220
8: MU 111 00 220 | 111 00 220
9: MU 111 00 220 | 111 00 220

>>> # add loci from another population
>>> pop2 = sim.Population(10, loci=2, lociPos=[0.1, 2.2], chromNames='chr3')
>>> pop.addLociFrom(pop2)
WARNING: Chromosome 'chr3' is merged to chromosome 'chr1'.
>>> pop.addLociFrom(pop2, byName=2)
>>> sim.dump(pop, genotype=False)
Ploidy: 2 (diploid)
Chromosomes:
1: chr1 (AUTOSOME, 5 loci)
   (0.1), (1), (2), (2.2), (3)
2: chr2 (AUTOSOME, 2 loci)
   rs2 (1), rs3 (2)
3: chr3 (AUTOSOME, 5 loci)
   (0.1), rs4 (1), (2.2), rs6 (3), rs8 (3.5)
population size: 10 (1 subpopulations with 10 Individuals)
Number of ancestral populations: 0

now exiting runScriptInteractively...

```

Download [addRemoveLoci.py](#)

4.3.9 Remove or extract individuals and subpopulations from a population

Functions `Population.removeIndividuals` and `Population.removeSubPops` remove selected individuals or groups of individuals from a population. Functions `Population.extractIndividuals` and `Population.extractSubPops` extract individuals and subpopulations from an existing population and form a new one.

Functions `removeIndividuals` and `extractIndividuals` could be used to remove or extract individuals from the present generation by indexes or from all ancestral generations by IDs or a Python filter function. This function should accept parameter `ind` or one or more information fields. `simuPOP` will pass individual for parameter `ind`, and values at specified information fields (`age` in this example) of each individual to this function. The present population structure will be kept, even if some subpopulations are left empty. For example, you could remove the first thirty individuals of a population using

```
pop.removeIndividuals(indexes=range(30))
```

or remove all individuals at age 20 or 30 using

```
pop.removeIndividuals(IDs=(20, 30), idField='age')
```

or remove all individuals with age between 20 and 30 using

```
pop.removeIndividuals(filter=lambda age: age >=20 and age <=30)
```

. In the last example, a Python lambda function is defined to avoid the definition of a named function.

Functions `removeSubPops` or `extractSubPops` could be used to remove or extract subpopulations, or groups of individuals defined by virtual subpopulations from a population. The latter case is very interesting because it could be used to remove or extract individuals with similar properties, such as all individuals between the ages 40 and 60, as demonstrated in Example *extract*.

Example: *Extract individuals*

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[200, 200], loci=[5, 5], infoFields='age')
>>> sim.initGenotype(pop, genotype=range(10))
>>> sim.initInfo(pop, lambda: random.randint(0,75), infoFields='age')
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', cutoff=[20, 60]))
>>> # remove individuals
>>> pop.removeIndividuals(indexes=range(0, 300, 10))
>>> print(pop.subPopSizes())
(180, 190)
>>> # remove individuals using IDs
>>> pop.setIndInfo([1, 2, 3, 4], field='age')
>>> pop.removeIndividuals(IDs=[2, 4], idField='age')
>>> # remove individuals using a filter function
>>> sim.initSex(pop)
>>> pop.removeIndividuals(filter=lambda ind: ind.sex() == sim.MALE)
>>> print([pop.individual(x).sex() for x in range(8)])
[2, 2, 2, 2, 2, 2, 2, 2]
>>> #
>>> # remove subpopulation
>>> pop.removeSubPops(1)
>>> print(pop.subPopSizes())
```

(continues on next page)

(continued from previous page)

```
(56,)
>>> # remove virtual subpopulation (people with age between 20 and 60)
>>> pop.removeSubPops([(0, 1)])
>>> print(pop.subPopSizes())
(56,)
>>> # extract another virtual subpopulation (people with age greater than 60)
>>> pop1 = pop.extractSubPops([(0, 2)])
>>> sim.dump(pop1, structure=False, max=10)
SubPopulation 0 (), 0 Individuals:

now exiting runScriptInteractively...
```

[Download extract.py](#)

4.3.10 Store arbitrary population information as population variables

Each simuPOP population has a Python dictionary that can be used to store arbitrary Python variables. These variables are usually used by various operators to share information between them. For example, the `Stat` operator calculates population statistics and stores the results in this Python dictionary. Other operators such as the `PyEval` and `TerminateIfread` from this dictionary and act upon its information.

simuPOP provides two functions, namely `Population.vars()` and `Population.dvars()` to access a population dictionary. These functions return the same dictionary object but `dvars()` returns a wrapper class so that you can access this dictionary as attributes. For example, `pop.vars()['alleleFreq'][0]` is equivalent to `pop.dvars().alleleFreq[0]`. Because dictionary `subPop[spID]` is frequently used by operators to store variables related to a particular (virtual) subpopulation, function `pop.vars(subPop)` is provided as a shortcut to `pop.vars()['subPop'][spID]`. Example *popVars* demonstrates how to set and access population variables.

Example: *population variables*

```
>>> import simuPOP as sim
>>> from pprint import pprint
>>> pop = sim.Population(100, loci=2)
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> print(pop.vars())      # No variable now
{}
>>> pop.dvars().myVar = 21
>>> print(pop.vars())
{'myVar': 21}
>>> sim.stat(pop, popSize=1, alleleFreq=0)
>>> # pprint prints in a less messy format
>>> pprint(pop.vars())
{'alleleFreq': {0: defdict({0: 0.275, 1: 0.725})},
 'alleleNum': {0: defdict({0: 55.0, 1: 145.0})},
 'myVar': 21,
 'popSize': 100,
 'subPopSize': [100]}
>>> # print number of allele 1 at locus 0
>>> print(pop.vars()['alleleNum'][0][1])
145.0
>>> # use the dvars() function to access dictionary keys as attributes
>>> print(pop.dvars().alleleNum[0][1])
145.0
>>> print(pop.dvars().alleleFreq[0])
```

(continues on next page)

(continued from previous page)

```
defdict({0: 0.275, 1: 0.725})

now exiting runScriptInteractively...
```

Download popVars.py

It is important to understand that this dictionary forms a **local namespace** in which Python expressions can be evaluated. This is the basis of how expression-based operators work. For example, the `PyEvaloperator` in example [simple_example](#) evaluates expression “`'%.2f\\t' % LD[0][1]`” in each population’s local namespace when it is applied to that population. This yields different results for different population because their LD values are different. In addition to Python expressions, Python statements can also be executed in the local namespace of a population, using the `stmts` parameter of the `PyEval` or `PyExec` operator. Example [expression](#) demonstrates the use of a `simuPOP` terminator, which terminates the evolution of a population when its expression is evaluated as `True`. Note that The `evolve()` function of this example does not specify how many generations to evolve so it will stop only after all replicates stop. The return value of this function indicates how many generations each replicate has evolved. This example also demonstrates how to run multiple replicates of an evolutionary process, which we will discuss in detail latter.

Example: *Expression evaluation in the local namespace of a population*

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=1), rep=5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme = sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.TerminateIf('len(alleleFreq[0]) == 1')
...     ]
... )
(129, 1540, 180, 247, 242)

now exiting runScriptInteractively...
```

Download expression.py

4.3.11 Save and load a population

`simuPOP` populations can be saved to and loaded from disk files using `Population.save(file)` member function and global function `loadPopulation`. **Virtual splitters are not saved** because they are considered as runtime definitions. Although files in any extension can be used, extension `.pop` is recommended. Example [savePop](#) demonstrates how to save and load a population in the native `simuPOP` format.

Example: *Save and load a population*

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=5, chromNames=['chrom1'])
>>> pop.dvars().name = 'my sim.Population'
>>> pop.save('sample.pop')
>>> pop1 = sim.loadPopulation('sample.pop')
>>> pop1.chromName(0)
'chrom1'
```

(continues on next page)

(continued from previous page)

```
>>> pop1.dvars().name
'my sim.Population'

now exiting runScriptInteractively...
```

Download savePop.py

The native simuPOP format is portable across different platforms but is not human readable and is not recognized by other applications. If you need to save a simuPOP population in a format that is recognizable by a particular software, you can use functions `importPopulation`, `export`, and `operator Exporter` if you would like to export populations during evolution. These functions are defined in module `simuPOP.utils`.

4.3.12 Import and export datasets in unsupported formats *

simuPOP provides a few utility functions to import and export populations in common formats such as GENEPOP, Phylip, and STRUCTURE (see chapter utility modules for details). If you need to import data from a file in a format that is not currently supported, you generally need to first scan the file for information such as number and names of chromosomes, loci, alleles, subpopulation, and individuals. After you create a population without genotype information from these parameters, you can scan the file for the second time and fill the population with genotypes and other information. Example `importData` demonstrates how to define a function to import from a file that is saved by function `saveCSV`.

Example: *Import a population from another file format*

```
>>> import simuPOP as sim
>>> def importData(filename):
...     'Read data from ``filename`` and create a population'
...     data = open(filename)
...     header = data.readline()
...     fields = header.split(',')
...     # columns 1, 3, 5, ..., without trailing '_1'
...     names = [fields[x].strip()[:-2] for x in range(1, len(fields), 2)]
...     popSize = 0
...     alleleNames = set()
...     for line in data.readlines():
...         # get all allele names
...         alleleNames |= set([x.strip() for x in line.split(',')[1:]])
...         popSize += 1
...     # create a population
...     alleleNames = list(alleleNames)
...     pop = sim.Population(size=popSize, loci=len(names), lociNames=names,
...         alleleNames=alleleNames)
...     # start from beginning of the file again
...     data.seek(0)
...     # discard the first line
...     data.readline()
...     for ind, line in zip(pop.individuals(), data.readlines()):
...         fields = [x.strip() for x in line.split(',')]
...         sex = sim.MALE if fields[0] == '1' else sim.FEMALE
...         ploidy0 = [alleleNames.index(fields[x]) for x in range(1, len(fields), 2)]
...         ploidy1 = [alleleNames.index(fields[x]) for x in range(2, len(fields), 2)]
...         ind.setGenotype(ploidy0, 0)
...         ind.setGenotype(ploidy1, 1)
...         ind.setSex(sex)
...     # close the file
```

(continues on next page)

(continued from previous page)

```

...     data.close()
...     return pop
...
>>> from simuPOP.utils import saveCSV
>>> pop = sim.Population(size=[10], loci=[3, 2], lociNames=['rs1', 'rs2', 'rs3', 'rs4',
↳ 'rs5'],
...     alleleNames=['A', 'B'])
>>> sim.initSex(pop)
>>> sim.initGenotype(pop, freq=[0.5, 0.5])
>>> # output sex but not affection status.
>>> saveCSV(pop, filename='sample.csv', affectionFormatter=None,
...     sexFormatter={sim.MALE:1, sim.FEMALE:2})
>>> # have a look at the file
>>> print(open('sample.csv').read())
sex, rs1_1, rs1_2, rs2_1, rs2_2, rs3_1, rs3_2, rs4_1, rs4_2, rs5_1, rs5_2
2, B, B, B, B, B, A, A, B, B, A
2, B, A, B, A, B, A, A, A, A, B
1, B, B, B, B, B, B, B, B, B, A
1, B, A, B, A, B, B, B, A, A, A
1, B, B, B, B, B, B, A, A, B, A
1, A, B, B, A, B, B, B, A, B, B
1, B, B, B, B, B, B, B, B, A, A
2, B, B, A, A, B, A, A, A, B, A
2, A, B, B, B, A, B, B, A, A, B
2, B, A, A, B, A, A, B, B, B, A

>>> pop1 = importData('sample.csv')
>>> sim.dump(pop1)
Ploidy: 2 (diploid)
Chromosomes:
1: (AUTOSOME, 5 loci)
   rs1 (1), rs2 (2), rs3 (3), rs4 (4), rs5 (5)
population size: 10 (1 subpopulations with 10 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 10 Individuals:
0: FU BBBAB | BBABA
1: FU BBBAA | AAAAB
2: MU BBBBB | BBBBA
3: MU BBBBA | AABAA
4: MU BBBAB | BBBA
5: MU ABBBB | BABAB
6: MU BBBBA | BBBBA
7: FU BABAB | BAAAA
8: FU ABABA | BBBAB
9: FU BAABB | ABABA

now exiting runScriptInteractively...
```

Download importData.py

Unless there are specific requirements in the order and labeling of individuals, exporting a simuPOP population is usually straightforward. Functions that are useful in such occasions include structural functions `Population.numSubPop()`, `Population.subPopName`, `Population.popSize()` and `Population.subPopSizes()`, and individual access functions `Population.individual()` and `Population.individuals()` and individual population access functions such as `Individual.allele()` and

Individual.info(). Function `saveFSTAT` in the cookbook module `fstatUtil` or `saveCSV` in module `simuPOP.utils` are good examples you can follow.

simuPOP Operators

simuPOP is large, consisting of more than 70 operators and various functions that covers all important aspects of genetic studies. These includes mutation (k -allele, stepwise, generalized stepwise), migration (arbitrary, can create new subpopulation), recombination (uniform or nonuniform), gene conversion, quantitative trait, selection, penetrance (single or multi-locus, hybrid), ascertainment (case-control, affected sibpairs, random), statistics calculation (allele, genotype, haplotype, heterozygote number and frequency; expected heterozygosity; bi-allelic and multi-allelic, and linkage disequilibrium measures; , and); pedigree tracing, visualization (using R or other Python modules). This chapter covers the basic and some not-so-basic usages of these operators, organized roughly by genetic factors.

5.1 Introduction to operators

Operators are objects that act on populations. There are two types of operators:

- **Operators that are applied to populations.** These operators are used in the `initOps`, `preOps`, `postOps` and `finalOps` parameters of the `evolve` function. The `initOps` operators are applied before an evolutionary process, the `preOps` operators are applied to the parental population at each generation before mating, the `postOps` operators are applied to the offspring population at each generation after mating, and the `finalOps` operators are applied after an evolutionary process. Examples of such operators include *MergeSubPops* to merge subpopulations and *StepwiseMutator* to mutate individuals using a stepwise mutation model.
- **Operators that are applied to individuals** (offspring) during mating. These operators are used in the `ops` parameter of a mating scheme. They are usually used to transmit genotype or other information from parents to offspring. Examples of such operators include *MendelianGenoTransmitter* that transmit parental genotype to offspring according to Mendelian laws and *ParentsTagger* that record the indexes of parents in the parental population to each offspring.

Some mutators could be applied both to populations and individuals. For example, an *IdTagger* could be applied to a whole population and assign an unique ID to all individuals, or to offspring during mating.

The following sections will introduce common features of all operators. The next chapter will explain all simuPOP operators in detail.

5.1.1 Apply operators to selected replicates and (virtual) subpopulations at selected generations

Operators are, by default, applied to all generations during an evolutionary process. This can be changed using the `begin`, `end`, `step` and `at` parameters. As their names indicate, these parameters control the starting generation (`begin`), ending generation (`end`), generations between two applicable generations (`step`), and an explicit list of applicable generations (`at`, a single generation number is also acceptable). Other parameters will be ignored if `at` is specified. It is worth noting that, if an evolutionary process has a pre-specified ending generation, negative generations are allowed. They are counted backward from the ending generation.

For example, if a simulator starts at generation 0, and the `evolve` function has parameter `gen=10`, the simulator will stop at the *beginning* of generation 10. Generation `-1` refers to generation 9, and generation `-2` refers to generation 8, and so on. Example *applicableGen* demonstrates how to set applicable generations of an operator. In this example, a population is initialized before evolution using an *InitGenotype* operator. allele frequency at locus 0 is calculated at generation 80, 90, but not 100 because the evolution stops at the beginning of generation 100. A *PyEval* operator outputs generation number and allele frequency at the end of generation 80 and 90. Another *PyEval* operator outputs similar information at generation 90 and 99, before and after mating. Note, however, because allele frequencies are only calculated twice, the pre-mating allele frequency at generation 90 is actually calculated at generation 80, and the allele frequencies display for generation 99 are calculated at generation 90. At the end of the evolution, the population is saved to a file using a *SavePopulation* operator.

Example: *Applicable generations of an operator.*

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[20])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.8, 0.2])
...     ],
...     preOps=[
...         sim.PyEval(r'''At the beginning of gen %d: allele Freq: %.2f\n' % (gen,
↪alleleFreq[0][0])",
...         at = [-10, -1])
...     ],
...     matingScheme = sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, begin=80, step=10),
...         sim.PyEval(r'''At the end of gen %d: allele freq: %.2f\n' % (gen,
↪alleleFreq[0][0])",
...         begin=80, step=10),
...         sim.PyEval(r'''At the end of gen %d: allele Freq: %.2f\n' % (gen,
↪alleleFreq[0][0])",
...         at = [-10, -1])
...     ],
...     finalOps=sim.SavePopulation(output='sample.pop'),
...     gen=100
... )
At the end of gen 80: allele freq: 0.92
At the beginning of gen 90: allele Freq: 0.92
At the end of gen 90: allele freq: 0.93
At the end of gen 90: allele Freq: 0.93
At the beginning of gen 99: allele Freq: 0.93
At the end of gen 99: allele Freq: 0.93
100

now exiting runScriptInteractively...
```

[Download applicableGen.py](#)

5.1.2 Applicable populations and (virtual) subpopulations

A simulator can evolve multiple replicates of a population simultaneously. Different operators can be applied to different replicates of this population. This allows side by side comparison between simulations.

Parameter `reps` is used to control which replicate(s) an operator can be applied to. This parameter can be a list of replicate numbers or a single replicate number. Negative index is allowed where `-1` refers to the last replicate. This technique has been widely used to produce table-like output where a *PyOutput* outputs a newline when it is applied to the last replicate of a simulator. Example *hybridOperator* demonstrates how to use this `reps` parameter. It is worth noting that negative indexes are *dynamic* indexes relative to number of active populations. For example, `rep=-1` will refer to a previous population if the last population has stopped evolving. Use a non-negative replicate number if this is not intended.

Example: *Apply operators to a subset of populations*

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=[20]), 5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.PyEval('gen', step=10, reps=0),
...         sim.PyEval(r"\t%.2f" % alleleFreq[0][0]", step=10, reps=(0, 2, -1)),
...         sim.PyOutput('\n', step=10, reps=-1)
...     ],
...     gen=30,
... )
0   0.23   0.22   0.29
10  0.15   0.23   0.21
20  0.04   0.07   0.10
(30, 30, 30, 30, 30)

now exiting runScriptInteractively...
```

[Download replicate.py](#)

An operator can also be applied to specified (virtual) subpopulations. For example, an initializer can be applied to male individuals in the first subpopulation, and everyone in the second subpopulation using parameter `subPops=[(0, 0), 1]`, if a virtual subpopulation is defined by individual sex. Generally speaking,

- `subPops=[]` applies the operator to all subpopulation. This is usually the default value of an operator.
- `subPops=[vsp1, vsp2, ...]` applies the operator all specified (virtual) subpopulations. (e.g. `subPops=[(0, 0), 1]`).
- `subPops=sp` is an abbreviation for `subPops=[sp]`. If `sp` is virtual, it has to be written as `[sp]` because `subPops=(0, 1)` is interpreted as two non-virtual subpopulation.

However, not all operators support this parameter, and even if they do, their interpretations of parameter input may vary. Please refer to documentation for individual operators in *the simuPOP reference manual* for details.

5.1.3 Dynamically determined loci (parameter `loci`) *

Many operators accept a parameter `loci` to specify the applicable loci. This parameter can be

- `ALL_AVAIL`: all available loci of the population to which the operator is applied.
- `[1, 2, 4, 5]`: A list of loci indexes. When the operator is applied to a population, it will be applied to the specified loci.
- `[('chr1', 5), ('chr1', 10), ('chr2', 5)]`: A list of chromosome position pairs. That is to say, when the operator is applied to a population, it will find loci at specified position of specified chromosome. Here chromosome names are names specified by parameter `chromNames` of the *Population* constructor. That is to say, the operator can be applied to all population with such chromosomes and loci at specified locations.
- `func`: A function with an optional parameter `pop`. When the operator is applied to a population, it will call this function, optionally pass the population to be applied to this function, and use its output as indexes of loci.

The last usage is very interesting because it allows the determination of loci according to population property. For example, Example *dynamicLoci* shows an example with a *MaSelector* that is applied to the locus with highest frequency at each generation by calling function `mostPopular`, which calculates allele frequency and pick the locus with highest allele frequency. This example looks silly, but the technique is very useful in simulating the introduction of disease loci by, for example, adding positive selection pressure to one of the chosen loci.

Example: *Natural selection with dynamically determined loci*

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=[10], infoFields='fitness')
>>>
>>> def mostPopular(pop):
...     sim.stat(pop, alleleFreq=sim.ALL_AVAIL)
...     freq = [pop.dvars().alleleFreq[x][1] for x in range(pop.totNumLoci())]
...     max_freq = max(freq)
...     pop.dvars().selLoci = (freq.index(max_freq), max_freq)
...     return [freq.index(max_freq)]
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.6, 0.4]),
...     ],
...     preOps=[
...         sim.MaSelector(fitness=[1, 0.9, 0.8], loci=mostPopular),
...         sim.PyEval(r"gen=%d, select against %d with frequency %.2f\n" % (gen,
↳ selLoci[0], selLoci[1])),
...     ],
...     matingScheme=sim.RandomMating(),
...     gen=10,
... )
gen=0, select against 6 with frequency 0.45
gen=1, select against 7 with frequency 0.46
gen=2, select against 2 with frequency 0.51
gen=3, select against 2 with frequency 0.48
gen=4, select against 2 with frequency 0.45
gen=5, select against 9 with frequency 0.45
gen=6, select against 3 with frequency 0.46
gen=7, select against 9 with frequency 0.44
gen=8, select against 7 with frequency 0.47
gen=9, select against 3 with frequency 0.44
```

(continues on next page)

(continued from previous page)

```
10
now exiting runScriptInteractively...
```

[Download dynamicLoci.py](#)

5.1.4 Write output of operators to one or more files

All operators we have seen, except for the *SavePopulation* operator in Example *applicableGen*, write their output to the standard output, namely your terminal window. However, it would be much easier for bookkeeping and further analysis if these output can be redirected to disk files. Parameter `output` is designed for this purpose.

Parameter `output` can take the following values:

- `' '` (an empty string): No output.
- `'>'`: Write to standard output.
- `'filename'` or `'>filename'`: Write the output to a file named `filename`. If multiple operators write to the same file, or if the same operator writes to the file file several times, only the last write operation will succeed.
- `'>>filename'`: Append the output to a file named `filename`. The file will be opened at the beginning of `evolve` function and closed at the end. An existing file will be cleared.
- `'>>>filename'`: This is similar to the `'>>'` form but the file will not be cleared at the beginning of the `evolve` function.
- `'!expr'`: `expr` is considered as a Python expression that will be evaluated at a population's local namespace whenever an output string is needed. For example, `'! '%d.txt' % gen'` would return `0.txt, 1.txt` etc at generation 0, 1,
- File handle of an opened file. Actually any python object with a `write` function.
- A Python function that can accept a string as its only parameter (`func(msg)`). When an operator outputs a message, this function will be called with this message.
- A `WithMode(output, 'b')` object with `output` being the any of the allowed output string or function. This object tells simuPOP that the output is opened in binary model so that it should output bytes instead of texts to it. This is mostly designed for Python 3 because file objects in Python 2 accepts string even if they are opened in binary mode.

Because a table output such as the one in Example *hybridOperator* is written by several operators, it is clear that all of them need to use the `'>>'` output format.

The *SavePopulation* operator in Example *applicableGen* write to file `sample.pop`. This works well if there is only one replicate but not so when the operator is applied to multiple populations. Only the last population will be saved successfully! In this case, the expression form of parameter `output` should be used.

The expression form of this parameter accepts a Python expression. Whenever a filename is needed, this expression is evaluated against the local namespace of the population it is applied to. Because the `evolve` function automatically sets variables `gen` and `rep` in a population's local namespace, such information can be used to produce an output string. Of course, any variable in this namespace can be used so you are not limited to these two variable.

Example *hybridOperator* demonstrates the use of these two parameters. In this example, a table is written to file `LD.txt` using `output='>>LD.txt'`. Similar operation to `output='R2.txt'` fails because only the last value is written to this file. The last operator writes output for each replicate to their respective output file such as `LD_0.txt`, using an expression that involves variable `rep`.

Example: Use the `output` and `outputExpr` parameters

(continued from previous page)

```

...     sim.InitSex(),
...     sim.InitGenotype(genotype=[1, 2, 2, 1])
... ],
... matingScheme = sim.RandomMating(ops=sim.Recombinator(rates=0.01)),
... postOps=[
...     sim.Stat(LD=[0, 1]),
...     sim.PyEval(r"'LD: %d, %.2f' % (gen, LD[0][1])", step=20,
...         output=logger.info), # send LD to console and a logfile
...     sim.PyEval(r"'R2: %d, %.2f' % (gen, R2[0][1])", step=20,
...         output=logger.debug), # send R2 only to a logfile
... ],
...     gen=100
... )
100
>>> print(open('simulation.log').read())
INFO: LD: 0, 0.25
DEBUG: R2: 0, 0.97
INFO: LD: 20, 0.20
DEBUG: R2: 20, 0.64
INFO: LD: 40, 0.18
DEBUG: R2: 40, 0.51
INFO: LD: 60, 0.12
DEBUG: R2: 60, 0.25
INFO: LD: 80, 0.10
DEBUG: R2: 80, 0.17

now exiting runScriptInteractively...

```

[Download outputFunc.py](#)

5.1.5 During-mating operators

All operators in Examples *applicableGen*, *replicate* and *output* are applied before or after mating. There is, however, a hidden during-mating operator that is called by *RandomMating()*. This operator is called *MendelianGenoTransmitter()* and is responsible for transmitting genotype from parents to offspring according to Mendel's laws. All pre-defined mating schemes (see Section *sec_Mating_Schemes*) use a special kind of during-mating operator to transmit genotypes. They are called **genotype transmitters** just to show the kind of task they perform. More during mating operators could be specified by replacing the default operator used in the *ops* parameter of a mating scheme (or an offspring generator if you are defining your own mating scheme).

Operators used in a mating scheme honor applicability parameters *begin*, *step*, *end*, *at* and *reps* although they do not support negative population and replicate indexes. It is therefore possible to apply different during-mating operators at different generations. For example, a *Recombinator* is used in Example *transmitter* to transmit parental genotypes to offspring after generation 30 while the *MendelianGenoTransmitter* is applied before that.

Example: *Genotype transmitters*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, loci=2)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 2, 2, 1])
...     ],

```

(continues on next page)

(continued from previous page)

```

...     matingScheme = sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(end=29),
...         sim.Recombinator(rates=0.01, begin=30),
...     ]),
...     postOps=[
...         sim.Stat(LD=[0, 1]),
...         sim.PyEval(r"gen %d, LD: %.2f\n" % (gen, LD[0][1])), step=20)
...     ],
...     gen=100
... )
gen 0, LD: 0.25
gen 20, LD: 0.25
gen 40, LD: 0.23
gen 60, LD: 0.19
gen 80, LD: 0.15
100

now exiting runScriptInteractively...
```

Download transmitter.py

During-mating operators can be applied to (virtual) subpopulations using parameter `subPops`, which **refers to (virtual) subpopulations in the offspring population**. Section [subsec_Pre_defined_genotype_transmitters](#) and [sec_Genotype_transmitters](#) list all genotype transmitters, Section [subsec_Customized_genotype_transmitter](#) demonstrates how to define your own genotype transmitter, Section [subsec_vspSelection](#) demonstrates the use of during-mating operator in virtual subpopulations.

5.1.6 Function form of an operator

Operators are usually applied to populations through a simulator but they can also be applied to a population directly. For example, it is possible to create an *InitGenotype* operator and apply to a population as follows:

```
InitGenotype(freq=[.3, .2, .5]).apply(pop)
```

Similarly, you can apply the hybrid penetrance model defined in Example [hybridOperator](#) to a population by

```
PyPenetrance(func=myPenetrance, loci=[10, 30, 50]).apply(pop)
```

This usage is used so often that it deserves some simplification. Equivalent functions are defined for most operators. For example, function `initGenotype` is defined for operator *InitGenotype* as follows

Example: *The function form of operator texttt{InitGenotype}*

```

>>> from simuPOP import InitGenotype, Population
>>> def initGenotype(pop, *args, **kwargs):
...     InitGenotype(*args, **kwargs).apply(pop)
...
>>> pop = Population(1000, loci=[2,3])
>>> initGenotype(pop, freq=[.2, .3, .5])

now exiting runScriptInteractively...
```

Download funcform.py

These functions are called function form of operators. Using these functions, the above two example can be written as


```
initGenotype(pop, freq=[.3, .2, .5])
```

and

```
pyPenetrance(pop, func=myPenetrance, loci=[10, 30, 50])
```

respectively. Note that applicability parameters such as `begin` and `end` can still be passed, but they are ignored by these functions.

Finally, it is worth noting that, if you have a function that manipulates population, you can make it an operator by wrapping it in a *PyOperator* so that it can be called repeatedly during evolution. For example, for a function `myFunc` that works on a population, you can define a wrapper function

```
def Func(pop):
    # call myFunc
    myFunc(pop)
    return True
```

which can then use it in a *PyOperator* as follows:

```
PyOperator(func=Func)
```

The wrapper function is not needed if `myFunc` returns `True` by itself. It can also be simplified to a lambda function

```
PyOperator(func=lambda pop: myFunc(pop) is None)
```

if you are certain that `myFunc` does not return any value (return `None`).

Note: Whereas output files specified by `'>'` are closed immediately after they are written, those specified by `'>>'` and `'>>>'` are not closed after the operator is applied to a population. This is not a problem when operators are used in a simulator because *Simulator.evolve* closes all files opened by operators, but can cause trouble when the operator is applied directly to a population. For example, multiple calls to `dump(pop, output='>>file')` will dump `pop` to `file` repeatedly but `file` will not be closed afterward. In this case, `closeOutput('file')` should be used to explicitly close the file.

5.2 Initialization

simuPOP provides three operators to initialize individual sex, information fields and genotype at the population level. A number of parameter are provided to cover most commonly used initialization scenarios. A Python operator can be used to initialize a population explicitly if none of the operators fits your need.

5.2.1 Initialize individual sex (operator *InitSex*)

Operator *InitSex*() and function `initSex()` initialize individual sex either randomly or using a given sequence. In the first case, individuals are assigned MALE or FEMALE with equal probability unless parameter *maleFreq* is used to specify the probability of having a male Individual. Alternatively, parameter *maleProp* can be used to specify exact proportions of male individuals so that you will have exactly 1000 males and 1000 females if you apply *InitSex*(`maleProp=0.5`) to a population of 2000 individuals.

Both parameters *maleFreq* and *maleProp* assigns individual sex randomly. If for some reason you need to specify individual sex explicitly, you could use a sequence of sex (MALE or FEMALE) to assign sex to individuals successively. The list will be reused if needed. If a list of (virtual) subpopulations are given, this operator will only initialize

individuals in these (virtual) subpopulations. Example *InitSex* demonstrates how to use two *InitSex* operators to initialize two subpopulations.

Example: *Initialize individual sex*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000, 1000])
>>> sim.initSex(pop, maleFreq=0.3, subPops=0)
>>> sim.initSex(pop, sex=[sim.MALE, sim.FEMALE, sim.FEMALE], subPops=1)
>>> sim.stat(pop, numOfMales=True, vars='numOfMales_sp')
>>> print(pop.dvars(0).numOfMales)
290
>>> print(pop.dvars(1).numOfMales)
334

now exiting runScriptInteractively...
```

[Download InitSex.py](#)

5.2.2 Initialize genotype (operator *InitGenotype*)

Operator *InitGenotype* (and its function form *initGenotype*) initializes individual genotype by allele frequency, allele proportion, haplotype frequency, haplotype proportions or a list of genotypes:

- By frequency of alleles. For example, *InitGenotype*(freq=(0, 0.2, 0.4, 0.2)) will assign allele 0, 1, 2, and 3 with probability 0, 0.2, 0.4 and 0.2 respectively.
- By proportions of alleles. For example, *InitGenotype*(prop=(0, 0.2, 0.4, 0.2)) will assign 400 allele 1, 800 allele 2 and 400 allele 3 to a diploid population with 800 individuals.
- By frequency of haplotypes. For example, *InitGenotype*(haplotypes=[[0, 0], [1,1], [0,1], [1,1]]) will assign four haplotypes with equal probabilities. *InitGenotype*(haplotypes=[[0, 0], [1,1], [0,1], [1,1]], freq=[0.2, 0.2, 0.3, 0.3]) will assign these haplotypes with different frequencies. If there are more than two loci, the haplotypes will be repeated.
- By frequency of haplotypes. For example, *InitGenotype*(haplotypes=[[0, 0], [1,1], [0,1], [1,1]], prop=[0.2, 0.2, 0.3, 0.3]) will assign four haplotypes with exact proportions.
- By a list of genotype. For example, *InitGenotype*(genotype=[1, 2, 2, 1]) will assign genotype 1, 2, 2, 1 repeatedly to a population. If individuals in this population has two homologous copies of a chromosome with two loci, this operator will assign haplotype 1, 2 to the first homologous copy of the chromosome, and 2, 1 to the second copy.
- By multiple allele frequencies or proportions returned by a function passed to parameter *freq* or *prop* (new in version 1.1.7). This function can accept parameters *loc*, *subPop* or *vsp* and returns locus, subpopulation or virtual subpopulation specific allele frequencies. For example, if you would like to initialize genotypes with random allele frequency, you can set *freq*=lambda : random.random() so that a new frequency is drawn from an uniform distribution for each new locus. Note that simuPOP expects the return value of this function to be a list of frequencies for alleles 0, 1, ..., but treats a single return value *x* as [*x*, 1-*x*] for simplicity.

Parameter *loci* and *ploidy* can be used to specify a subset of loci and homologous sets of chromosomes to initialize, and parameter *subPops* can be used to specify subsets of individuals to initialize. Example *InitGenotype* demonstrates how to use these the *InitGenotype* operator, including examples on how to define and use virtual subpopulations to initialize individual genotype by sex or by proportion.

Example: *Initialize individual genotype*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000, 3000], loci=[5, 7])
>>> # by allele frequency
>>> def printFreq(pop, loci):
...     sim.stat(pop, alleleFreq=loci)
...     print(' '.join(['{:3f}'.format(pop.dvars().alleleFreq[x][0]) for x in_
↳ loci]))
...
>>> sim.initGenotype(pop, freq=[.4, .6])
>>> sim.dump(pop, max=6, structure=False)
SubPopulation 0 (), 2000 Individuals:
  0: MU 11000 0011111 | 11111 0101110
  1: MU 00000 1111111 | 11101 1111001
  2: MU 10111 0111100 | 01111 1011111
  3: MU 11011 1101010 | 11010 1011111
  4: MU 11011 0011010 | 10011 1001110
  5: MU 00001 1010011 | 11111 1111110
SubPopulation 1 (), 3000 Individuals:
2000: MU 10011 0010100 | 01001 0011010

>>> printFreq(pop, range(5))
0.397, 0.404, 0.400, 0.402, 0.406
>>> # by proportion
>>> sim.initGenotype(pop, prop=[0.4, 0.6])
>>> printFreq(pop, range(5))
0.400, 0.400, 0.400, 0.400, 0.400
>>> # by haplotype frequency
>>> sim.initGenotype(pop, freq=[.4, .6], haplotypes=[[1, 1, 0, 1], [0, 0, 1]])
>>> sim.dump(pop, max=6, structure=False)
SubPopulation 0 (), 2000 Individuals:
  0: MU 11011 1011101 | 00100 1001001
  1: MU 11011 1011101 | 11011 1011101
  2: MU 00100 1001001 | 00100 1001001
  3: MU 00100 1001001 | 00100 1001001
  4: MU 11011 1011101 | 11011 1011101
  5: MU 00100 1001001 | 11011 1011101
SubPopulation 1 (), 3000 Individuals:
2000: MU 00100 1001001 | 00100 1001001

>>> printFreq(pop, range(5))
0.597, 0.597, 0.403, 0.597, 0.597
>>> # by haplotype proportion
>>> sim.initGenotype(pop, prop=[0.4, 0.6], haplotypes=[[1, 1, 0], [0, 0, 1, 1]])
>>> printFreq(pop, range(5))
0.600, 0.600, 0.400, 0.000, 0.600
>>> # by genotype
>>> pop = sim.Population(size=[2, 3], loci=[5, 7])
>>> sim.initGenotype(pop, genotype=[1]*5 + [2]*7 + [3]*5 + [4]*7)
>>> sim.dump(pop, structure=False)
SubPopulation 0 (), 2 Individuals:
  0: MU 11111 2222222 | 33333 4444444
  1: MU 11111 2222222 | 33333 4444444
SubPopulation 1 (), 3 Individuals:
  2: MU 11111 2222222 | 33333 4444444
  3: MU 11111 2222222 | 33333 4444444
  4: MU 11111 2222222 | 33333 4444444

```

(continues on next page)

(continued from previous page)

```

>>> #
>>> # use virtual subpopulation
>>> pop = sim.Population(size=[2000, 3000], loci=[5, 7])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> sim.initSex(pop)
>>> sim.initGenotype(pop, genotype=range(10), loci=range(5))
>>> # initialize all males
>>> sim.initGenotype(pop, genotype=[2]*7, loci=range(5, 12),
...     subPops=[(0, 0), (1, 0)])
>>> # assign genotype by proportions
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.4, 0.6]))
>>> sim.initGenotype(pop, freq=[0.2, 0.8], subPops=[(0,0)])
>>> sim.initGenotype(pop, freq=[0.5, 0.5], subPops=[(0,1)])
>>> #
>>> # initialize by random allele frequency
>>> import random
>>> sim.initGenotype(pop, freq=lambda : random.random())
>>> printFreq(pop, range(5))
0.580, 0.239, 0.100, 0.576, 0.674
>>> # initialize with loci specific frequency. here
>>> # lambda loc: 0.01*loc is equivalent to
>>> # lambda loc: [0.01*loc, 1-0.01*loc]
>>> sim.initGenotype(pop,
...     freq=lambda loc: 0.01*loc)
>>> printFreq(pop, range(5))
0.000, 0.009, 0.018, 0.029, 0.041
>>> # initialize with VSP-specific frequency
>>> sim.initGenotype(pop,
...     freq=lambda vsp: [[0.2, 0.8], [0.5, 0.5]][vsp[1]],
...     subPops=[(0, 0), (0, 1)])
>>>

now exiting runScriptInteractively...

```

[Download InitGenotype.py](#)

5.2.3 Initialize information fields (operator `InitInfo`)

Operator `InitInfo` and its function form `initInfo` initialize one or more information fields of all individuals or Individuals in selected (virtual) subpopulations using either a list of values or a Python function. If a value or a list of value is given, it will be used repeatedly to assign values of specified information fields of all applicable individuals. For example, `initInfo(pop, values=1, infoFields='x')` will assign value 1 to information field `x` of all individuals, and

```
initInfo(pop, values=[1, 2, 3], infoFields='x', subPops=[(0,1)])
```

will assign values 1, 2, 3, 1, 2, 3... to information field `x` of individuals in the second virtual subpopulation of subpopulation 0.

The `values` parameter also accepts a Python function. This feature is usually used to assign random values to an information field. For example, `values=random.random` would assign a random value between 0 and 1. If a function takes parameters, a lambda function can be used. For example,

```
initInfo(pop, lambda : random.randint(2, 5), infoFields=['x', 'y'])
```

assigns random integers between 2 and 5 to information fields `x` and `y` of all individuals in `pop`. Example *InitInfo* demonstrates these usages.

Example: *initialize information fields*

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=[5], loci=[2], infoFields=['sex', 'age'])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> sim.initSex(pop)
>>> sim.initInfo(pop, 0, subPops=[(0,0)], infoFields='sex')
>>> sim.initInfo(pop, 1, subPops=[(0,1)], infoFields='sex')
>>> sim.initInfo(pop, lambda: random.randint(20, 70), infoFields='age')
>>> sim.dump(pop, structure=False)
SubPopulation 0 (), 5 Individuals:
  0: FU 00 | 00 | 1 39
  1: FU 00 | 00 | 1 29
  2: MU 00 | 00 | 0 68
  3: MU 00 | 00 | 0 50
  4: MU 00 | 00 | 0 21

now exiting runScriptInteractively...
```

[Download InitInfo.py](#)

5.3 Expressions and statements

5.3.1 Output a Python string (operator `PyOutput`)

Operator *PyOutput* is a simple operator that prints a Python string when it is applied to a population. It is commonly used to print the progress of a simulation (e.g. *PyOutput*('start migration\\n', at=200)) or output separators to beautify outputs from *PyEval* outputs (e.g. *PyOutput*('\\n', rep=-1)).

5.3.2 Execute Python statements (operator `PyExec`)

Operator *PyExec* executes Python statements in a population's local namespace when it is applied to that population. This operator is designed to execute short Python statements but multiple statements separated by newline characters are allowed.

Example *PyExec* uses two *PyExec* operators to create and use a variable `traj` in each population's local namespace. The first operator initialize this variable as an empty list. During evolution, the frequency of allele 1 at locus 0 is calculated (operator *Stat*) and appended to this variable (operator *PyExec*). The result is a trajectory of allele frequencies during evolution.

Example: *Execute Python statements during evolution*

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=1),
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8]),
```

(continues on next page)

(continued from previous page)

```

...     sim.PyExec('traj=[]')
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyExec('traj.append(alleleFreq[0][1])'),
...     ],
...     gen=5
... )
(5, 5)
>>> # print Trajectory
>>> print(', '.join(['%.3f' % x for x in simu.dvars(0).traj]))
0.775, 0.790, 0.760, 0.750, 0.750

now exiting runScriptInteractively...
```

[Download PyExec.py](#)

5.3.3 Evaluate and output Python expressions (operator `PyEval`)

Operator `PyEval` evaluate a given Python expression in a population's local namespace and output its return value. This operator has been widely used (e.g. Example [simple_example](#), [ancestralPop](#), [applicableGen](#) and [output](#)) to output statistics of populations and report progress.

Two additional features of this operator may become handy from time to time. First, an optional Python statements (parameter `stmts`) can be specified which will be executed before the expression is evaluated. Second, the population being applied can be exposed in its own namespace as a variable (parameter `exposePop`). This makes it possible to access properties of a population other than its variables. Example `PyEval` demonstrates both features. In this example, two statements are executed to count the number of unique parents in an offspring population and save them as variables `numFather` and `numMother`. The operator outputs these two variables along with a generation number.

Example: Evaluate a expression and statements in a population's local namespace.

```

>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1,
...     infoFields=['mother_idx', 'father_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.ParentsTagger(),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r'"gen %d, #father %d, #mother %d\n" \
...             ' % (gen, numFather, numMother)',
...             stmts="numFather = len(set(pop.indInfo('father_idx')))\n"
...                 "numMother = len(set(pop.indInfo('mother_idx')))",
...             exposePop='pop')
...     ],
...     gen=3
... )
gen 0, #father 439, #mother 433
gen 1, #father 433, #mother 432
gen 2, #father 449, #mother 420
```

(continues on next page)

(continued from previous page)

```
3
now exiting runScriptInteractively...
```

[Download PyEval.py](#)

Note that the function form of this operator (*pyEval*) returns the result of the expression rather than writing it to an output.

5.3.4 Expression and statement involving individual information fields (operator *InfoEval* and *InfoExec*) *

Operators *PyEval* and *PyExec* work at the population level, using the local namespace of populations. Operator *InfoEval* and *InfoExec*, on the contrary, work at the individual level, using individual information fields (and population variables) as variables. In this case, individual information fields are copied to the population namespace one by one before expression or statements are executed for each individual. Optionally, the individual object can be exposed to these namespace using a user-specified name (parameter *exposeInd*). Individual information fields will be updated if the value of these fields are changed.

Operator *InfoEval* evaluates an expression and outputs its value. Operator *InfoExec* executes one or more statements and does not produce any output. Operator *InfoEval* is usually used to output individual information fields and properties in batch mode. It is faster and sometimes easier to use than corresponding for loop plus individual level operations. For example

- *InfoEval*(`r' '%.2f\\t' ' % a'`) outputs the value of information field *a* for all individuals, separated by tabs.
- *InfoEval*(`'ind.sexChar()', exposeInd='ind'`) outputs the sex of all individuals using an exposed individual object *ind*.
- *InfoEval*(`'a+b**2'`) outputs for information fields and for all individuals.

Example *InfoEval* demonstrates the use of this operator.

Example: *Evaluate expressions using individual information fields*

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(20, loci=1, infoFields='a')
>>> pop.setVirtualSplitter(sim.InfoSplitter('a', cutoff=[3]))
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> pop.setIndInfo([random.randint(2, 5) for x in range(20)], 'a')
>>> sim.infoEval(pop, 'a', subPops=[(0, 0)]);print(' ')
2.02.02.02.0
>>> sim.infoEval(pop, 'ind.allele(0, 0)', exposeInd='ind');print(' ')
110111111111100111111
>>> # use sim.population variables
>>> pop.dvars().b = 5
>>> sim.infoEval(pop, "%d " % (a+b));print(' ')
8 9 10 8 9 10 8 9 10 10 9 7 9 7 9 7 9 7 9 8
now exiting runScriptInteractively...
```

[Download InfoEval.py](#)

Operator *InfoExec* is usually used to set individual information fields. For example

- *InfoExec*(`'age += 1'`) increases the age of all individuals by one.

- `InfoExec('risk = 2 if packPerYear > 10 else 1.5')` sets information field `risk` to 2 if `packPerYear` is greater than 10, and 1.5 otherwise. Note that conditional expression is only available for Python version 2.5 or later.
- `InfoExec('a = b*c')` sets the value of information field `a` to the product of `b` and `c`.

Example *InfoExec* demonstrates the use of this operator, using its function form `infoExec`.

Example: *Execute statements using individual information fields*

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=1, infoFields=['a', 'b', 'c'])
>>> sim.initSex(pop)
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> sim.infoExec(pop, 'a=1')
>>> print(pop.indInfo('a')[:10])
(1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0)
>>> sim.infoExec(pop, 'b=ind.sex()', exposeInd='ind')
>>> print(pop.indInfo('b')[:10])
(2.0, 2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 2.0)
>>> sim.infoExec(pop, 'c=a+b')
>>> print(pop.indInfo('c')[:10])
(3.0, 3.0, 2.0, 2.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0)
>>> pop.dvars().d = 5
>>> sim.infoExec(pop, 'c+=d')
>>> print(pop.indInfo('c')[:10])
(8.0, 8.0, 7.0, 7.0, 7.0, 7.0, 7.0, 8.0, 8.0, 8.0)
>>> # the operator can update population variable as well
>>> sim.infoExec(pop, 'd+=c*c')
>>> print(pop.dvars().d)
5835.0

now exiting runScriptInteractively...
```

[Download InfoExec.py](#)

Note that a statement can also be specified for operator *InfoEval*, which will be executed before an expression is evaluated.

5.3.5 Using functions in external modules in simuPOP expressions and statements

All simuPOP expressions and statements are evaluated in a population's local namespace, which is a dictionary with no access to external modules. If you would like to use external modules (e.g. functions from the `random` module), you will have to import them to the namespace explicitly, using something like

```
exec('import random', pop.vars(), pop.vars())
```

before you evolve the population.

Example *outputByInterval* demonstrates the application of this technique. This example imports the `time` module in the population's local namespace and set `init_time` and `last_time` before evolution. During evolution, an *IfElse* operator is used to output the status of the simulation for every 5 seconds using expression `time.time() - last_time > 5`. `last_time` is reset using the *PyExec* operator. The evolution will last 20 seconds and be terminated by the Terminator with expression `time.time() - init_time > 20`.

Example: *Write the status of an evolutionary process every 10 seconds*


```

>>> import simuPOP as sim
>>> import time
>>> pop = sim.Population(1000, loci=10)
>>> pop.dvars().init_time = time.time()
>>> pop.dvars().last_time = time.time()
>>> exec('import time', pop.vars(), pop.vars())
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.IfElse('time.time() - last_time > 5', [
...             sim.PyEval(r'"Gen: %d\n" % gen'),
...             sim.PyExec('last_time = time.time()'),
...             ]),
...         sim.TerminateIf('time.time() - init_time > 20')
...     ]
... )
Gen: 5043
Gen: 9971
Gen: 14997
19925
>>>

now exiting runScriptInteractively...

```

[Download outputByInterval.py](#)

5.4 Demographic changes

A mating scheme controls the size of an offspring generation using parameter `subPopSize`. This parameter has been described in detail in section [subsec_offspring_size](#). In summary,

- The subpopulation sizes of the offspring generation will be the same as the parental generation if `subPopSize` is not set.
- The offspring generation will have a fixed size if `subPopSize` is set to a number (no subpopulation) or a list of subpopulation sizes.
- The subpopulation sizes of an offspring generation will be determined by the return value of a demographic function if `subPopSize` is set to such a function (a function that returns subpopulation sizes at each generation).

Note: Parameter `subPopSize` only controls subpopulation sizes of an offspring generation immediately after it is generated. population or subpopulation sizes could be changed by other operators.

During mating, a mating scheme goes through each parental subpopulation and populates its corresponding offspring subpopulation. This implies that

- Parental and offspring populations should have the same number of subpopulations.
- Mating happens strictly within each subpopulation.

This section will introduce several operators that allow you to move individuals across the boundary of subpopulations (migration), and change the number of subpopulations during evolution (split and merge). Please refer to [subsec_offspring_size](#) (control the size of the offspring generation section of chapter mating scheme) for more details. For more advanced demographic models, please refer to the [simuPOP.demography](#) module.

5.4.1 Migration (operator *Migrator*)

Migration by probability

Operator *Migrator* (and its function form *migrate*) migrates individuals from one subpopulation to another. The key parameters are

- *from* subpopulations (parameter *subPops*). A list of subpopulations from which individuals migrate. Default to all subpopulations.
- *to* subpopulations (parameter *toSubPops*). A list of subpopulations to which individuals migrate. Default to all subpopulations. **A new subpopulation ID can be specified to create a new subpopulation from migrants.**
- A migration rate matrix (parameter *rate*). A by matrix (a nested list in Python) that specifies migration rate from each source to each destination subpopulation. That is to say, specifies migration rate from *to* . Needless to say, and are determined by the number of *from* and *to* subpopulations.

Example *migrateByProb* demonstrate the use of a *Migrator* to migrate individuals between three subpopulations. Note that

- Operator *Migrator* relies on an information field *migrate_to* (configurable) to record destination subpopulation of each individual so this information field needs to be added to a population before migration.
- Migration rates to subpopulation themselves are determined automatically so they can be left unspecified.

Example: Migration by probability

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*3, infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[
...         [0, 0.1, 0.1],
...         [0, 0, 0.1],
...         [0, 0.1, 0]
...     ]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval('subPopSize'),
...         sim.PyOutput('\n')
...     ],
...     gen = 5
... )
[762, 1108, 1130]
[601, 1175, 1224]
[490, 1233, 1277]
[395, 1282, 1323]
[320, 1300, 1380]
5
now exiting runScriptInteractively...
```

[Download migrateByProb.py](#)

Migration by proportion and counts

Migration rate specified in the rate parameter in Example *migrateByProb* is interpreted as probabilities. That is to say, a migration rate is interpreted as the probability at which any individual in subpopulation migrates to subpopulation .

The exact number of migrants are randomly distributed.

If you would like to specify exactly how many migrants migrate from a subpopulation to another, you can specify parameter `mode` of operator *Migrator* to `BY_PROPORTION` or `BY_COUNTS`. The `BY_PROPORTION` mode interpret as proportion of individuals who will migrate from subpopulation to so the number of migrant will be exactly `subPopSize(m)`. In the `BY_COUNTS` mode, is interpreted as number of migrants, regardless the size of subpopulation. Example *migrateByPropAndCount* demonstrates these two migration modes, as well as the use of parameters `subPops` and `toSubPops`.

Example: *Migration by proportion and count*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*3, infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[[0.1], [0.2]],
...                           mode=sim.BY_PROPORTION,
...                           subPops=[1, 2],
...                           toSubPops=[3]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval('subPopSize'),
...         sim.PyOutput('\n')
...     ],
...     gen = 5
... )
[1000, 900, 800, 300]
[1000, 810, 640, 550]
[1000, 729, 512, 759]
[1000, 657, 410, 933]
[1000, 592, 328, 1080]
5
>>> #
>>> pop.evolve(
...     preOps=sim.Migrator(rate=[[50, 50], [100, 50]],
...                           mode=sim.BY_COUNTS,
...                           subPops=[3, 2],
...                           toSubPops=[2, 1]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval('subPopSize'),
...         sim.PyOutput('\n')
...     ],
...     gen = 5
... )
[1000, 692, 328, 980]
[1000, 792, 328, 880]
[1000, 892, 328, 780]
[1000, 992, 328, 680]
[1000, 1092, 328, 580]
5
now exiting runScriptInteractively...
```

[Download migrateByPropAndCount.py](#)

Theoretical migration models

To facilitate the use of widely used theoretical migration models, a few functions are defined in module `simuPOP.demography.subsec_Predefined_migration_models`. These functions generate migration matrixes that can be plugged in to the `Migrator` operator.

migrate from virtual subpopulations *

Under a realistic eco-social settings, individuals in a subpopulation rarely have the same probability to migrate. Genetic evidence has shown that female has a higher migrate rate than male in humans, perhaps due to migration patterns related to inter-population marriages. Such sex-biased migration also happens in other large migration events such as slave trade.

It is easy to simulate most of such complex migration models by migrating from virtual subpopulations. For example, if you define virtual subpopulations by sex, you can specify different migration rates for males and females and control the proportion of males among migrants, by specifying virtual subpopulations in parameter `subPops`. Parameter `toSubPops` does not accept virtual subpopulations because you cannot, for example, migrate to females in a subpopulation.

Example `migrateVSP` demonstrate a sex-biased migration model where males dominate migrants from subpopulation 0. To avoid confusing, this example uses the proportion migration mode. At the beginning of the first generation, there are 500 males and 500 females in each subpopulation. A 10% male migration rate and 5% female migration rate leads to 50 male migrants and 25 female migrants. Subpopulation sizes and number of males in each subpopulation before mating are therefore:

- Subpopulation 0: male 500-50, female 500-25, total 925
- Subpopulation 1: male 500+50, female 500+25, total 1075

Note that the unspecified `to` subpopulations are subpopulation 0 and 1, which cannot be virtual.

Example: Migration from virtual subpopulations

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*2, infoFields='migrate_to')
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     # 500 males and 500 females
...     initOps=sim.InitSex(sex=[sim.MALE, sim.FEMALE]),
...     preOps=[
...         sim.Migrator(rate=[
...             [0, 0.10],
...             [0, 0.05],
...         ],
...         mode = sim.BY_PROPORTION,
...         subPops=[(0, 0), (0, 1)]),
...     sim.Stat(popSize=True, numOfMales=True, vars='numOfMales_sp'),
...     sim.PyEval(r"%d/%d\t%d/%d\n" % (subPop[0]['numOfMales'], subPopSize[0], "
...         "subPop[1]['numOfMales'], subPopSize[1])),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True, numOfMales=True, vars='numOfMales_sp'),
...         sim.PyEval(r"%d/%d\t%d/%d\n" % (subPop[0]['numOfMales'], subPopSize[0], "
...             "subPop[1]['numOfMales'], subPopSize[1])),
...     ],
...     gen = 2
... )
```

(continues on next page)

(continued from previous page)

```

450/925      550/1075
426/925      520/1075
384/859      562/1141
425/859      582/1141
2
now exiting runScriptInteractively...
```

[Download migrateVSP.py](#)

Arbitrary migration models **

If none of the described migration methods fits your need, you can always resort to manual migration. One such example is when you need to mimick an existing evolutionary scenario so you know exactly which subpopulation each individual will migrate to.

Manual migration is actually very easy. All you need to do is specifying the destination subpopulation of all individuals in the *from* subpopulations (parameter `subPops`), using an information field (usually `migrate_to`). You can then call the *Migrator* using `mode=BY_IND_INFO`. Example *manualMigration* shows how to manually move individuals around. This example uses the function form of *Migrator*. You usually need to use a Python operator to set destination subpopulations if you would like to manually migrate individuals during an evolutionary process.

Example: Manual migration

```

>>> import simuPOP as sim
>>> pop = sim.Population([10]*2, infoFields='migrate_to')
>>> pop.setIndInfo([0, 1, 2, 3]*5, 'migrate_to')
>>> sim.migrate(pop, mode=sim.BY_IND_INFO)
>>> pop.subPopSizes()
(5, 5, 5, 5)
now exiting runScriptInteractively...
```

[Download manualMigration.py](#)

Note: individuals with an invalid destination subpopulation ID (e.g. an negative number) will be discarded silently. Although not recommended, this feature can be used to remove individuals from a subpopulation.

5.4.2 Migration using backward migration matrix (operator *BackwardMigrator*)

Backward migration matrices are widely used in theoretical population genetics and coalescent based simulations. Instead of specifying the probability of migrating from one subpopulation to another (namely how migration happens), such matrices specify the probability that individuals in a subpopulation originate from others (namely the result of migration). *simuPOP* simulates such models by converting backward migration matrices to forward ones using the theory described below. Due to the limit of such models, *simuPOP* cannot simulate migration from/to virtual subpopulations, creation of new subpopulation, different source and destination subpopulations, and will generate an error if the conversion process fails.

To explain the differences between forward and backward migration matrices, let us assume that there are subpopulations with population sizes s_i , and a forward migration matrix

where p_{ij} is the probability that an individual will migrate from subpopulation j to i . After migration happens, subpopulation sizes are changed to n_i , and the origin of individuals in each subpopulation can be described by the backward migration matrix

where q_{ij} is the probability that an individual in subpopulation i originates from subpopulation j .

These quantities can be derived from original population sizes and the forward migration matrix. That is to say, the size of new subpopulation i is the sum of all migrants to this subpopulation

and the size of the original population is the sum of all migrants from this subpopulation

and the composition of subpopulation i (e.g. individuals originate from subpopulation j) is

In matrix form, these formulas can be written as

and

Therefore, given a backward migration matrix and current population size n , we can derive a forward migration matrix using

and

Note that n is always true if p is symmetric and (equal subpopulation size) so simuPOP will use p directly in this case. Also note that p might not be invertible and n might be invalid (e.g. negative population size or forward migration rate) for given p and n . simuPOP will terminate with an error message in these cases.

The following example *backwardMigration* demonstrates how to use a backward migration matrix to perform migration. It initializes all individuals with indexes of subpopulations they belong to before migration and calculates the percent of individuals from each source population using a PyOperator with function `originOfInds`. The so-called overseeded backward migration matrix is similar to specified migration matrix despite of stochastic effects. This example also uses `turnOnDebug` function to let the operator print the expected subpopulation size (`spSize`) and calculate forward migration matrix (`B_sp`) at each generation, which, as expected, vary from generation to generation.

Example: *Migration using a backward migration matrix*

```
>>> import simuPOP as sim
>>> sim.turnOnDebug('DBG_MIGRATOR')
>>> pop = sim.Population(size=[10000, 5000, 8000], infoFields=['migrate_to', 'migrate_
↳ from'])
>>> def originOfInds(pop):
...     print('Observed backward migration matrix at generation {}'.format(pop.
↳ dvars().gen))
...     for sp in range(pop.numSubPop()):
...         # get source subpop for all individuals in subpopulation i
...         origins = pop.indInfo('migrate_from', sp)
...         spSize = pop.subPopSize(sp)
...         B_sp = [origins.count(j) * 1.0 / spSize for j in range(pop.numSubPop())]
...         print('      ' + ', '.join(['{:3f}'.format(x) for x in B_sp]))
...     return True
...
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=
...         # mark the source subpopulation of each individual
...         [sim.InitInfo(i, subPops=i, infoFields='migrate_from') for i in range(3)]_
↳ + [
...         # perform migration
...         sim.BackwardMigrator(rate=[
...             [0, 0.04, 0.02],
...             [0.05, 0, 0.02],
...             [0.02, 0.01, 0]
```

(continues on next page)

(continued from previous page)

```

...     ]),
...     # calculate and print observed backward migration matrix
...     sim.PyOperator(func=originOfInds),
...     # calculate population size
...     sim.Stat(popSize=True),
...     # and print it
...     sim.PyEval(r'"Pop size after migration: {}\\n".format(", ".join([str(x)
↳for x in subPopSize]))'),
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 5
... )
Expected next population size is 10211.4, 4851.8, 7936.84
Forward migration matrix is 0.959867, 0.024259, 0.0158737, 0.0816908, 0.902435, 0.
↳0158737, 0.0255284, 0.0121295, 0.962342
Observed backward migration matrix at generation 0
    0.939, 0.040, 0.021
    0.051, 0.927, 0.022
    0.020, 0.010, 0.969
Pop size after migration: 10218, 4859, 7923
Expected next population size is 10453.6, 4690.64, 7855.79
Forward migration matrix is 0.961671, 0.0229529, 0.0153764, 0.0860553, 0.897777, 0.
↳0161675, 0.0263879, 0.0118406, 0.961772
Observed backward migration matrix at generation 1
    0.942, 0.038, 0.020
    0.049, 0.932, 0.020
    0.023, 0.010, 0.968
Pop size after migration: 10417, 4706, 7877
Expected next population size is 10675.5, 4517.1, 7807.37
Forward migration matrix is 0.963329, 0.0216814, 0.0149897, 0.0907397, 0.89267, 0.
↳0165902, 0.0271056, 0.0114691, 0.961425
Observed backward migration matrix at generation 2
    0.942, 0.039, 0.020
    0.048, 0.930, 0.022
    0.020, 0.010, 0.970
Pop size after migration: 10660, 4536, 7804
Expected next population size is 10946, 4323.5, 7730.53
Forward migration matrix is 0.965217, 0.0202791, 0.0145038, 0.0965253, 0.886432, 0.
↳0170426, 0.0280522, 0.0110802, 0.960868
Observed backward migration matrix at generation 3
    0.940, 0.040, 0.020
    0.050, 0.930, 0.020
    0.020, 0.011, 0.969
Pop size after migration: 10942, 4321, 7737
Expected next population size is 11260.4, 4079.55, 7660
Forward migration matrix is 0.967357, 0.0186417, 0.0140011, 0.104239, 0.878033, 0.
↳0177274, 0.0291081, 0.0105456, 0.960346
Observed backward migration matrix at generation 4
    0.937, 0.043, 0.021
    0.046, 0.933, 0.021
    0.019, 0.009, 0.972
Pop size after migration: 11331, 4042, 7627
5

now exiting runScriptInteractively...

```

[Download backwardMigrate.py](#)

5.4.3 Split subpopulations (operators `SplitSubPops`)

Operator `SplitSubPops` splits one or more subpopulations into finer subpopulations. It can be used to simulate populations that originate from the same founder population. For example, a population of size 1000 in Example *split-BySize* is split into three subpopulations of sizes 300, 300 and 400 respectively, after evolving as a single population for two generations.

Example: *Split subpopulations by size*

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000)
>>> pop.evolve(
...     preOps=[
...         sim.SplitSubPops(subPops=0, sizes=[300, 300, 400], at=2),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(),
...     gen = 4
... )
Gen 0:      [1000]
Gen 1:      [1000]
Gen 2:      [300, 300, 400]
Gen 3:      [300, 300, 400]
4
now exiting runScriptInteractively...
```

[Download splitBySize.py](#)

Operator `SplitSubPops` splits a subpopulation by sizes of the resulting subpopulations. It is often easier to do so with proportions. In addition, if a demographic function is used, you should make sure that the number of subpopulations will be the same before and after mating at any generation. One way of doing this is to apply a `SplitSubPops` operator at the right generation. Example *splitByProp* demonstrates such an evolutionary scenario. However, it is often easier to split the population in the demographic function in such case (see section *subsec_Advanced_demo_func* for details).

Example: *Split subpopulations by proportion*

```
>>> import simuPOP as sim
>>> def demo(gen, pop):
...     if gen < 2:
...         return 1000 + 100 * gen
...     else:
...         return [x + 50 * gen for x in pop.subPopSizes()]
...
>>> pop = sim.Population(1000)
>>> pop.evolve(
...     preOps=[
...         sim.SplitSubPops(subPops=0, proportions=[.5]*2, at=2),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(subPopSize=demo),
...     gen = 4
... )
Gen 0:      [1000]
Gen 1:      [1000]
```

(continues on next page)

(continued from previous page)

```

Gen 2:      [550, 550]
Gen 3:      [650, 650]
4
now exiting runScriptInteractively...

```

Download `splitByProp.py`

Either by *sizes* or by *proportions*, individuals in a subpopulation are divided randomly. It is, however, also possible to split subpopulations according to individual information fields. In this case, individuals with different values at a given information field will be split into different subpopulations. This is demonstrated in Example *splitByInfo* where the function form of operator *SplitSubPops* is used.

Example: *Split subpopulations by individual information field*

```

>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population([1000]*3, subPopNames=['a', 'b', 'c'], infoFields='x')
>>> pop.setIndInfo([random.randint(0, 3) for x in range(1000)], 'x')
>>> print(pop.subPopSizes())
(1000, 1000, 1000)
>>> print(pop.subPopNames())
('a', 'b', 'c')
>>> sim.splitSubPops(pop, subPops=[0, 2], infoFields=['x'])
>>> print(pop.subPopSizes())
(243, 244, 262, 251, 1000, 243, 244, 262, 251)
>>> print(pop.subPopNames())
('a', 'a', 'a', 'a', 'b', 'c', 'c', 'c', 'c')
now exiting runScriptInteractively...

```

Download `splitByInfo.py`

5.4.4 Merge subpopulations (operator *MergeSubPops*)

Operator *MergeSubPops* merges specified subpopulations into a single subpopulation. This operator can be used to simulate admixed populations where two or more subpopulations merged into one subpopulation and continue to evolve for a few generations. Example *MergeSubPops* simulates such an evolutionary scenario. A demographic model could be added similar to Example *splitByProp*.

Example: *Merge multiple subpopulations into a single subpopulation*

```

>>> import simuPOP as sim
>>> pop = sim.Population([500]*2)
>>> pop.evolve(
...     preOps=[
...         sim.MergeSubPops(subPops=[0, 1], at=3),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(),
...     gen = 5
... )
Gen 0:      [500, 500]
Gen 1:      [500, 500]
Gen 2:      [500, 500]

```

(continues on next page)

(continued from previous page)

```

Gen 3:      [1000]
Gen 4:      [1000]
5
now exiting runScriptInteractively...

```

[Download MergeSubPops.py](#)

5.4.5 Resize subpopulations (operator `ResizeSubPops`)

Whenever possible, it is recommended that subpopulation sizes are changed naturally, namely through the population of an offspring generation. However, it is sometimes desired to change the size of a population forcefully. Examples of such applications include immediate expansion of a small population before evolution, and the simulation of sudden population size change caused by natural disaster. By default, new individuals created by such sudden population expansion get their genotype from existing individuals. Example *ResizeSubPops* shows a scenario where two subpopulations expand instantly at generation 3.

Example: *Resize subpopulation sizes*

```

>>> import simuPOP as sim
>>> pop = sim.Population([500]*2)
>>> pop.evolve(
...     preOps=[
...         sim.ResizeSubPops(proportions=(1.5, 2), at=3),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(),
...     gen = 5
... )
Gen 0:      [500, 500]
Gen 1:      [500, 500]
Gen 2:      [500, 500]
Gen 3:      [750, 1000]
Gen 4:      [750, 1000]
5
now exiting runScriptInteractively...

```

[Download ResizeSubPops.py](#)

5.4.6 Time-dependent migration rate

In evolutionary scenarios with complex demographic models, number of subpopulations and migration rate might change from generation to generation. For example, if one of the subpopulations is split into two, the migration matrix has to be changed to accommodate increased number of subpopulations.

If there are a limited number of demographic changes and a few number of pre-determined migration matrices. You can use a number of `Migrators` that are applied at different generations. For example, you can use the following operators to apply the first migration scheme during first ten generations (0, ..., 9), and the second migration scheme during the rest of the evolutionary process:

```

preOps=[
    Migrator(rate=M1, end=9),

```

(continues on next page)

(continued from previous page)

```

Migrator(rate=M2, begin=10),
]

```

If changes of demographics are frequent or stochastic so that migration matrices can only be determined programmatically, it is easier to use a *PyOperator* to migrate populations using the function form of a *Migrator*. This is demonstrated in Example *varyingMigr* where migration matrixes are computed dynamically due to random split of subpopulations.

Example: *Varying migration rate*

```

>>> import simuPOP as sim
>>>
>>> from simuPOP.utils import migrIslandRates
>>> import random
>>>
>>> def demo(pop):
...     # this function randomly split populations
...     numSP = pop.numSubPop()
...     if random.random() > 0.3:
...         pop.splitSubPop(random.randint(0, numSP-1), [0.5, 0.5])
...     return pop.subPopSizes()
...
>>> def migr(pop):
...     numSP = pop.numSubPop()
...     sim.migrate(pop, migrIslandRates(0.01, numSP))
...     return True
...
>>> pop = sim.Population(10000, infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.PyOperator(func=migr),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     gen = 5
... )
Gen 0:      [10000]
Gen 1:      [4982, 5018]
Gen 2:      [2495, 2505, 5000]
Gen 3:      [2509, 2517, 4974]
Gen 4:      [2512, 2512, 4976]
5
now exiting runScriptInteractively...

```

[Download VaryingMigr.py](#)

5.5 Genotype transmitters

5.5.1 Generic genotype transmitters (operators `GenoTransmitter`, `CloneGenoTransmitter`, `MendelianGenoTransmitter`, `SelfingGenoTransmitter`, `HaplodiploidGenoTransmitter`, and `MitochondrialGenoTransmitter`)*

A number of during-mating operators are defined to transmit genotype from parent(s) to offspring. They are rarely used or even seen directly because they are used as genotype transmitters of mating schemes.

- *GenoTransmitter*: This genotype transmitter is usually used by customized genotype transmitters because it provides some utility functions that are more efficient than their Pythonic counterparts.
- *CloneGenoTransmitter*: Copy all genotype on non-customized chromosomes from a parent to an offspring. It also copies parental sex to the offspring because sex can be genotype determined. This genotype transmitter is used by mating scheme *CloneMating*. This genotype transmitter can be applied to populations of **any ploidy** type. If you would like to copy part of the chromosomes, or customized chromosomes, a parameter `chroms` could be used to specify chromosomes to copy.
- *MendelianGenoTransmitter*: Copy genotypes from two parents (a male and a female) to an offspring following Mendel's laws, used by mating scheme *RandomMating*. This genotype transmitter can only be applied to **diploid** populations.
- *SelfingGenoTransmitter*: Copy genotypes from one parent to an offspring using self-fertilization, used by mating scheme *SelfMating*. This genotype transmitter can only be applied to **diploid** populations.
- *HaplodiploidGenoTransmitter*: Set genotype to male and female offspring differently in a haplodiploid population, used by mating scheme *HaplodiploidMating*. This genotype transmitter can only be applied to **haplodiploid** populations.
- *MitochondrialGenoTransmitter*: Treat a single mitochondrial chromosome, or all customized chromosomes, or specified chromosomes as mitochondrial chromosomes and transmit maternal mitochondrial chromosomes randomly to an offspring. This genotype transmitter can be applied to populations of **any ploidy** type. It transmits the first homologous copy of chromosomes maternally and clears alleles on other homologous copies of chromosomes of an offspring.

5.5.2 Recombination (Operator `Recombinator`)

The generic genotype transmitters do not handle genetic recombination. A genotype transmitter *Recombinator* is provided for such purposes, and can be used with *RandomMating* and *SelfMating* (replace *MendelianGenoTransmitter* and *SelfingGenoTransmitter* used in these mating schemes).

Recombination rate is implemented **between adjacent markers**. There can be only one recombination event between adjacent markers no matter how far apart they are located on a chromosome. In practise, a *Recombinator* goes along chromosomes and determine, between each adjacent loci, whether or not a recombination happens.

Recombination rates could be specified in the following ways:

1. If a single recombination rate is specified through parameter `rates`, it will be the recombination rate between all adjacent loci, regardless of loci position.
2. If recombination happens only after certain loci, you can specify these loci using parameter `loci`. For example,

```
Recombinator(rates=0.1, loci=[2, 5])
```

recombines a chromosome only **after** loci 2 (between 2 and 3) and 5 (between 5 and 6).

3. If parameter `loci` is given with a list of loci, different recombination rate can be given to each of them. The two lists should have the same length. For example

```
Recombinator(rates=[0.1, 0.05], loci=[2, 5])
```

uses two different recombination rates after loci 2 and 5.

4. If parameter `loci` is not given (default to `loci=ALL_AVAIL`) but a list of recombination rates is assigned, the rates will be assigned to each locus. The length of parameter `rates` should equal to total number of loci but the recombination rates for the locus at the end of each chromosome will be ignored (assumed to be 0.5). For example

```
Recombinator(rates=[0.1]*5 + [0.2]*5)
```

uses two different recombination rates for two chromosomes with 5 loci.

5. If recombination rates vary across your chromosomes, a long list of `rate` and `loci` may be needed to specify recombination rates one by one. An alternative method is to specify a **recombination intensity**. Recombination rate between two adjacent loci is calculated as the product of this intensity and distance between them. For example, if you apply operator

```
Recombinator(intensity=0.1)
```

to a population

```
Population(size=100, loci=[4], lociPos=[0.1, 0.2, 0.4, 0.8])
```

The recombination rates between adjacent markers will be 0.1×0.1 , 0.1×0.2 and 0.1×0.4 respectively.

Example: Genetic recombination at all and selected loci

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(size=[1000], loci=[100]),
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*100 + [1]*100)
...     ],
...     matingScheme=sim.RandomMating(ops = [
...         sim.Recombinator(rates=0.01, reps=0),
...         sim.Recombinator(rates=[0.01]*10, loci=range(50, 60), reps=1),
...     ]),
...     postOps=[
...         sim.Stat(LD=[[40, 55], [60, 70]]),
...         sim.PyEval(r'%d:\t%.3f\t%.3f\t' % (rep, LD_prime[40][55], LD_
↪prime[60][70])),
...         sim.PyOutput('\n', reps=-1)
...     ],
...     gen = 5
... )
0:  0.741  0.806  1:      0.904  1.000
0:  0.658  0.715  1:      0.882  1.000
0:  0.491  0.668  1:      0.843  1.000
0:  0.435  0.610  1:      0.818  1.000
0:  0.383  0.567  1:      0.763  1.000
(5, 5)

now exiting runScriptInteractively...
```

Download [recRate.py](#)

Example [recRate](#) demonstrates how to specify recombination rates for all loci or for specified loci. In this example, two replicates of a population are evolved, subject to two different Recombinators. The first Recombinator applies the same recombination rate between all adjacent loci, and the second Recombinator recombines only after loci 50 - 59. Because there is no recombination event between loci 60 and 70 for the second replicate, linkage disequilibrium values between these two loci does not decrease as what happens in the first replicate.

Example: *Genetic recombination rates specified by intensity*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000], loci=3, lociPos=[0, 1, 1.1])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*3 + [1]*3)
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(intensity=0.01)),
...     postOps=[
...         sim.Stat(LD=[[0, 1], [1, 2]]),
...         sim.PyEval(r'%.3f\t%.3f\n' % (LD_prime[0][1], LD_prime[1][2])), step=10)
...     ],
...     gen = 50
... )
0.988      0.998
0.912      0.996
0.836      0.991
0.896      0.982
0.814      0.991
50
now exiting runScriptInteractively...
```

Download [recIntensity.py](#)

Example [recIntensity](#) demonstrates the use of the `intensity` parameter. In this example, the distances between the first two loci and the latter two loci are 1 and 0.1 respectively. This leads recombination rates 0.01 and 0.001 respectively with a recombination intensity 0.01. Consequently, LD between the first two loci decay much faster than the latter two.

If more advanced recombination model is desired, a customized genotype transmitter can be used. For example, Example [sexSpecificRec](#) uses two Recombinators to implement sex-specific recombination.

Note: Both loci positions and recombination intensity are unitless. You can assume different unit for loci position and recombination intensity as long as the resulting recombination rate makes sense.

5.5.3 Gene conversion (Operator [Recombinator](#)) *

simuPOP uses the Holliday junction model to simulate gene conversion. This model treats recombination and conversion as a unified process. The key features of this model is

- Two (out of four) chromatids pair and a single strand cut is made in each chromatid
- Strand exchange takes place between the chromatids
- Ligation occurs yielding two completely intact DNA molecules
- Branch migration occurs, giving regions of heteroduplex DNA

- Resolution of the Holliday junction gives two DNA molecules with heteroduplex DNA. Depending upon how the holliday junction is resolved, we either observe no exchange of flanking markers, or an exchange of flanking markers. The former forms a conversion event, which can be considered as a double recombination.

In practise, gene conversion can be considered as a double recombination event. That is to say, when a recombination event happens, it has certain probability to trigger a second recombination event along the chromosome. The distance between the two locations where recombination events happen is the tract length of this conversion event.

The probability at which gene conversion happens, and how tract length is determined is specify using parameter `convMode` of a `Recombinator`. This parameter can be

- `NoConversion` No gene conversion. (default)
- `(NUM_MARKERS, prob, N)` Convert a fixed number `N` of markers at probability `prob`.
- `(TRACT_LENGTH, prob, N)` Convert a fixed length `N` of chromosome regions at probability `prob`. This can be used when markers are not equally spaced on chromosomes.
- `(GEOMETRIC_DISTRIBUTION, prob, p)` When a conversion event happens at probability `prob`, convert a random number of markers, with a geometric distribution with parameter `p`.
- `(EXPONENTIAL_DISTRIBUTION, prob, p)` When a conversion event happens at probability `prob`, convert a random length of chromosome region, using an exponential distribution with parameter `p`.

Note that

- If tract length is determined by length (`TractLength` or `ExponentialDistribution`), the starting point of the flanking region is uniformly distributed between marker and , if the recombination happens at marker . That is to say, it is possible that no marker is converted with a positive tract length.
- A conversion event will act like a recombination event if its flanking region exceeds the end of a chromosome, or if another recombination event happens before the end of the flanking region.

Example [conversion](#) compares two Recombinators. The first Recombinator is a regular Recombinator that recombine between loci 50 and 51. The second Recombinator is a conversion operator because every recombination event will become a conversion event (`prob=1`). Because a second recombination event will surely happen between loci 60 and 61, there will be either no or double recombination events between loci 40, 70. LD between these two loci therefore does not decrease, although LD between locus 55 and these two loci will decay.

Example: Gene conversion

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(size=[1000], loci=[100]),
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*100 + [1]*100)
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.Recombinator(rates=0.01, loci=50, reps=0),
...         sim.Recombinator(rates=0.01, loci=50, reps=1, convMode=(sim.NUM_MARKERS,
↪1, 10)),
...     ]),
...     postOps=[
...         sim.Stat(LD=[[40, 55], [40, 70]]),
...         sim.PyEval(r'%d:\t%.3f\t%.3f\t' % (rep, LD_prime[40][55], LD_
↪prime[40][70])),
...         sim.PyOutput('\n', reps=-1)
...     ],
...     gen = 5
```

(continues on next page)

(continued from previous page)

```

... )
0:  0.988  0.988  1:      0.980  1.000
0:  0.982  0.982  1:      0.982  1.000
0:  0.982  0.982  1:      0.974  1.000
0:  0.974  0.974  1:      0.954  1.000
0:  0.960  0.960  1:      0.940  1.000
(5, 5)

now exiting runScriptInteractively...

```

[Download conversion.py](#)

5.5.4 Tracking all recombination events **

To understand the evolutionary history of a simulated population, it is sometimes needed to track down all ancestral recombination events. In order to do that, you will first need to give an unique ID to each individual so that you could make sense of the dumped recombination events. Although this is routinely done using operator *IdTagger* (see example *IdTagger* for details), it is a little tricky here because you need to place the during- mating *IdTagger* before a *Recombinator* in the ops parameter of a mating scheme so that offspring ID could be set and outputted correctly.

After setting the name of the ID field (usually `ind_id`) to the `infoField` parameter of a *Recombinator*, it can dump a list of recombination events (loci after which recombination events happened) for each set of homologous chromosomes of an offspring. Each line is in the format of

```
offspringID parentID startingPloidy rec1 rec2 ....
```

Example *trackRec* gives an example how the output looks like.

Example: *Tracking all recombination events*

```

>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[1000, 2000], infoFields='ind_id')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...     ],
...     matingScheme=sim.RandomMating(ops = [
...         sim.IdTagger(),
...         sim.Recombinator(rates=0.001, output='>>rec.log', infoFields='ind_id')]),
...     gen = 5
... )
5
>>> rec = open('rec.log')
>>> # print the first three lines of the log file
>>> print(''.join(rec.readlines()[:4]))
1001 642 0 381 999 1490
1001 250 1 908 999 1315 2134
1002 847 1 999
1002 91 0 975 999 1245 2546

now exiting runScriptInteractively...

```

[Download trackRec.py](#)

5.6 Mutation

A mutator (a mutation operator) mutates alleles at certain loci from one allele to another. Because alleles are simple non-negative numbers that can be interpreted as nucleotides, codons, sequences of nucleotides or even genetic deletions, appropriate mutation models have to be chosen for different types of loci. Please refer to Section [sec_Genotypic_structure](#) for a few examples.

A mutator will mutate alleles at all loci unless parameter `loci` is used to specify a subset of loci. Different mutators have different concepts and forms of mutation rates. If a mutator accepts only a single mutation rate (which can be in the form of a list or a matrix), it uses parameter `rate` and applies the same mutation rate to all loci. If a mutator accepts a list of mutation rates (each of which is a single number), it uses parameter `rates` and applies different mutation rates to different loci if multiple loci are specified. Note that parameter `rates` also accepts single form inputs (e.g. `rates=0.01`) in which case the same mutation rate will be applied to all loci.

5.6.1 Mutation models specified by rate matrixes (**MatrixMutator**)

A mutation model can be defined as a **mutation rate matrix** where is the probability that an allele mutates to per generation per locus. Although mathematical formulation of are sometimes unscaled, simuPOP assumes for all and requires such rate matrixes in the specification of a mutation model. of such a matrix are ignored because they are automatically calculated from .

A *MatrixMutator* is defined to mutate between alleles 0, 1, ..., according to a given rate matrix. Conceptually speaking, this mutator goes through each mutable allele and mutates it to allele according to probabilities , . Most alleles will be kept intact because mutations usually happen at low probability (with close to 1). For example, Example *MatrixMutator* simulates a locus with 3 alleles. Because the rate at which allele 2 mutats to alleles 0 and 1 is higher than the rate alleles 0 and 2 mutate to allele 2, the frequency of allele 2 decreases over time.

Example: *General mutator specified by a mutation rate matrix*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.3, 0.5])
...     ],
...     preOps=sim.MatrixMutator(rate = [
...         [0, 1e-5, 1e-5],
...         [1e-4, 0, 1e-4],
...         [1e-3, 1e-3, 0]
...     ]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=100),
...         sim.PyEval(r"', '.join(['%.3f' % alleleFreq[0][x] for x in range(3)]) +
↪ '\n"',
...         step=100),
...     ],
...     gen=1000
... )
0.192, 0.302, 0.505
0.241, 0.292, 0.467
0.328, 0.273, 0.399
0.270, 0.322, 0.408
0.312, 0.412, 0.276
0.330, 0.344, 0.327
```

(continues on next page)

(continued from previous page)

```

0.332, 0.424, 0.244
0.426, 0.372, 0.201
0.413, 0.384, 0.203
0.395, 0.408, 0.198
1000

now exiting runScriptInteractively...
```

[Download MatrixMutator.py](#)

Note: Alleles other than 0, 1, ..., \$n-1\$ will not be mutated because their mutation rates are undefined. A warning message will be displayed for this case when debugging code `DBG_WARNING` is turned on.

5.6.2 k-allele mutation model (`KAlleleMutator`)

A *k*-allele model assumes alleles at a locus and mutate between them using rate matrix

The only parameter is the mutation rate, which is the rate at which an allele mutates to any other allele with equal probability.

This mutation model is a special case of the `MatrixMutator` but a specialized `KAlleleMutator` is recommended because it provides better performance, especially when *k* is large. In addition, this operator allows different mutation rates at different loci. When *k* is not specified, it is assumed to be the number of allowed alleles (e.g. 2 for binary modules). Example `KAlleleMutator` demonstrates the use of this operator where parameters `rate` and `loci` are used to specify different mutation rates for different loci. Because this operator treats all alleles equally, all alleles will have the same allele frequency in the long run.

Example: A *k*-allele mutation model

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1*3)
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.KAlleleMutator(k=5, rates=[1e-2, 1e-3], loci=[0, 1]),
...         sim.Stat(alleleFreq=range(3), step=100),
...         sim.PyEval(r' ".join(['%.3f' % alleleFreq[x][0] for x in range(3)]) +
↳ '\n',
...         step=100),
...     ],
...     gen=500
... )
0.991, 0.999, 1.000
0.368, 0.918, 1.000
0.300, 0.815, 1.000
0.257, 0.639, 1.000
0.209, 0.573, 1.000
500

now exiting runScriptInteractively...
```

[Download KAlleleMutator.py](#)

Note: If alleles \$k\$ and higher exist in the population, they will not be mutated because their mutation rates are undefined. A warning message will be displayed for this case when debugging code `DBG_WARNING` is turned on.

5.6.3 Diallelic mutation models (`SNPMutator`)

`MatrixMutator` and `KAlleleMutator` are general purpose mutators in the sense that they do not assume a type for the mutated alleles. This and the following sections describe mutation models for specific types of alleles.

If there are only two alleles at a locus, a diallelic mutation model should be used. Because single nucleotide polymorphisms (SNPs) are the most widely available diallelic markers, a `SNPMutator` is provided to mutate such markers using a mutate rate matrix

Despite of its name, this mutator can be used in many theoretical models assuming and . If , mutations will be directional. Example `SNPMutator` applies such a directional mutaton model to two loci, but with a purifying selection applied to the first locus. Because of the selection pressure, the frequency of allele 1 at the first locus does not increase indefinitely as allele 1 at the second locus.

Example: *A diallelic directional mutation model*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=[1, 1], lociNames=['A', 'B'],
...   infoFields='fitness')
>>> pop.evolve(
...   initOps=sim.InitSex(),
...   preOps=[
...     sim.SNPMutator(u=0.001),
...     sim.MaSelector(loci='A', fitness=[1, 0.99, 0.98]),
...   ],
...   matingScheme=sim.RandomMating(),
...   postOps=[
...     sim.Stat(alleleFreq=['A', 'B'], step=100),
...     sim.PyEval(r"%3f\t%3f\n" % (alleleFreq[0][1], alleleFreq[1][1])),
...     step=100),
...   ],
...   gen=500
... )
0.001      0.001
0.077      0.087
0.099      0.192
0.099      0.300
0.085      0.400
500
now exiting runScriptInteractively...
```

[Download SNPMutator.py](#)

5.6.4 Nucleotide mutation models (`AcgtMutator`)

Mutations in these models assume alleles 0, 1, 2, 3 as nucleotides A, C, G, and T. The operator is named `AcgtMutator` to remind you the alphabetic order of these nucleotides. This mutation model is specified by a rate matrix

which is determined by 12 parameters. However, several simpler models with fewer parameters can be used. In addition to parameters shared by all mutation operators, a nucleotide mutator is specified by a parameter list and a model name. For example:

```
AcgtMutator(rate=[1e-5, 0.5], model='K80')
```

specifies a nucleotide mutator using Kimura's 2-parameter model with and . Because multiple parameters could be involved for a particular mutation model, **the definition of a mutation rate and other parameters are model dependent and may vary with different mathematical representation of the models.**

The names and acceptable parameters of acceptable models are listed below:

1. Jukes and Cantor 1969 model: model='JC69', rate=[]

The Jukes and Cantor model is similar to a -allele model but its definition of is different. More specifically, when a mutation event happens at rate , an allele will have equal probability to mutate to any of the 4 allelic states.

2. Kimura's 2-parameter 1980 model: model='K80', rate=[,]

Kimura's model distinguishes transitions (, and namely and with probability) and transversions (others) with probability . It would be a Jukes and Cantor model if .

3. Felsenstein 1981 model: model='F81', rate=[, , ,].

This model assumes different base frequencies but the same probabilities for transitions and transversions. is calculated from , and .

4. Hasegawa, Kishino and Yano 1985 model: model='HKY85', rate=[, , , ,]

This model replaces 1/4 frequency used in the Kimura's 2-parameter model with nucleotide-specific frequencies.

5. Tamura 1992 model: model='T92', rate=[,]

This model is a HKY85 model with and ,

6. Tamura and Nei 1993 model: model='TN93', rate=[, , , , ,]

This model extends the HKY1985 model by distinguishing transitions (namely) and transitions () with different .

7. Generalized time reversible model: model='GTR', rate=[, , , , , , ,]

The generalized time reversible model is the most general neutral, independent, finite-sites, time-reversible model possible. It is specified by six parameters and base frequencies. Its rate matrix is defined as

8. General model: model='general' (default), rate=[, , , , , , , , , , , ,].

This is the most general model with 12 parameters:

It is not surprising that all other models are implemented as special cases of this model.

Example *AcgtMutator* applies a Kimura's 2-parameter mutation model to a population with a single nucleotide marker.

Example: *A Kimura's 2 parameter mutation model*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1,
...     alleleNames=['A', 'C', 'G', 'T'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.1, .1, .1, .7])
```

(continues on next page)

(continued from previous page)

```

...     ],
...     matingScheme=sim.RandomMating(),
...     preOps=[
...         sim.AcgtMutator(rate=[1e-4, 0.5], model='K80'),
...         sim.Stat(alleleFreq=0, step=100),
...         sim.PyEval(r"','.join(['%.3f' % alleleFreq[0][x] for x in range(4)]) +
↳ '\n'",
...         step=100),
...     ],
...     gen=500
... )
0.093, 0.101, 0.094, 0.712
0.142, 0.073, 0.084, 0.701
0.135, 0.160, 0.083, 0.623
0.230, 0.128, 0.013, 0.628
0.293, 0.189, 0.008, 0.510
500

now exiting runScriptInteractively...

```

[Download AcgtMutator.py](#)

5.6.5 Mutation model for microsatellite markers (*StepwiseMutator*)

The **stepwise mutation model** (SMM) was proposed by Ohta1973 to model the mutation of Variable Number Tandem Repeat (VNTR), which consists of tandem repeat of sequences. VNTR markers consisting of short sequences (e.g. 5 basepair or less) are also called microsatellite markers. A mutation event of a VNTR marker either increase or decrease the number of repeats, as a result of slipped-strand mispairing or unequal sister chromatid exchange and genetic recombination.

A *StepwiseMutator* assumes that alleles at a locus are the number of tandem repeats and mutates them by increasing or decreasing the number of repeats during a mutation event. By adjusting parameters `incProb`, `maxAllele` and `mutStep`, this operator can be used to simulate the standard neutral stepwise mutation model and a number of **generalized stepwise mutation models**. For example, Example *StepwiseMutator* uses two *StepwiseMutator* to mutate two microsatellite markers, using a standard and a generalized model where a geometric distribution is used to determine the number of steps.

Example: *A standard and a generalized stepwise mutation model*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=[1, 1])
>>> pop.evolve(
...     # all start from allele 50
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq= [0]*50 + [1])
...     ],
...     matingScheme=sim.RandomMating(),
...     preOps=[
...         sim.StepwiseMutator(rates=1e-3, loci=0),
...         sim.StepwiseMutator(rates=1e-3, incProb=0.6, loci=1,
...             mutStep=(sim.GEOMETRIC_DISTRIBUTION, 0.2)),
...     ],
...     gen=100
... )

```

(continues on next page)

(continued from previous page)

```

100
>>> # count the average number tandem repeats at both loci
>>> cnt0 = cnt1 = 0
>>> for ind in pop.individuals():
...     cnt0 += ind.allele(0, 0) + ind.allele(0, 1)
...     cnt1 += ind.allele(1, 0) + ind.allele(1, 1)
...
>>> print('Average number of repeats at two loci are %.2f and %.2f.' % \
...       (cnt0/2000., cnt1/2000.))
Average number of repeats at two loci are 50.03 and 49.70.

now exiting runScriptInteractively...

```

[Download StepwiseMutator.py](#)

5.6.6 Simulating arbitrary mutation models using a hybrid mutator (PyMutator)*

A hybrid mutator *PyMutator* mutates random alleles at selected loci (parameter *loci*), replicates (parameter *loci*), subpopulations (parameter *subPop*) with specified mutation rate (parameter *rate*). Instead of mutating the alleles by itself, it passes the alleles to a user-defined function and use it return values as the mutated alleles. Arbitrary mutation models could be implemented using this operator.

Example *PyMutator* applies a simple mutation model where an allele is increased by a random number between 1 and 5 when it is mutated. Two different mutation rates are used for two different loci so average alleles at these two loci are different.

Example: *A hybrid mutation model*

```

>>> import simuPOP as sim
>>> import random
>>> def incAllele(allele):
...     return allele + random.randint(1, 5)
...
>>> pop = sim.Population(size=1000, loci=[20])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(),
...     postOps=sim.PyMutator(func=incAllele, rates=[1e-4, 1e-3],
...                           loci=[2, 10]),
...     gen = 1000
... )
1000
>>> # count the average number tandem repeats at both loci
>>> def avgAllele(pop, loc):
...     ret = 0
...     for ind in pop.individuals():
...         ret += ind.allele(loc, 0) + ind.allele(loc, 1)
...     return ret / (pop.popSize() * 2.)
...
>>> print('Average number of repeats at two loci are %.2f and %.2f.' % \
...       (avgAllele(pop, 2), avgAllele(pop, 10)))
Average number of repeats at two loci are 0.01 and 2.19.

now exiting runScriptInteractively...

```

[Download PyMutator.py](#)

5.6.7 Mixed mutation models (*MixedMutator*) **

Mixed mutation models are sometimes used to model real data. For example, a -allele model can be used to explain extremely large or small number of tandem repeats at a microsatellite marker which are hard to justify using a standard stepwise mutation model. A mixed mutation model would apply two or more mutation models at pre-specified probabilities.

A *MixedMutator* is constructed by a list of mutators and their respective probabilities. It accepts regular mutator parameters such as rates, loci, subPops, mapIn and mapOut and mutates alleles at specified rate. When a mutation event happens, it calls one of the mutators to mutate the allele. For example, Example *MixedMutator* applies a mixture of -allele model and stepwise model to mutate a microsatellite model.

Example: *A mixed k-allele and stepwise mutation model*

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=[1, 1])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[50, 50])
...     ],
...     preOps=[
...         # the first locus uses a pure stepwise mutation model
...         sim.StepwiseMutator(rates=0.001, loci=0),
...         # the second locus uses a mixed model
...         sim.MixedMutator(rates=0.001, loci=1, mutators=[
...             sim.KAlleleMutator(rates=1, k=100),
...             sim.StepwiseMutator(rates=1)
...         ], prob=[0.1, 0.9])),
...     matingScheme=sim.RandomMating(),
...     gen = 20
... )
20
>>> # what alleles are there?
>>> geno0 = []
>>> geno1 = []
>>> for ind in pop.individuals():
...     geno0.extend([ind.allele(0, 0), ind.allele(0, 1)])
...     geno1.extend([ind.allele(1, 0), ind.allele(1, 1)])
...
>>> print('Locus 0 has alleles', ', '.join([str(x) for x in set(geno0)]))
Locus 0 has alleles 49, 50, 51
>>> print('Locus 1 has alleles', ', '.join([str(x) for x in set(geno1)]))
Locus 1 has alleles 67, 49, 50, 51, 88

now exiting runScriptInteractively...
```

[Download MixedMutator.py](#)

When a mutation event happens, mutators in Example *MixedMutator* mutate the allele with probability (mutation rate) 1. If different mutation rates are specified, the overall mutation rates would be the product of mutation rate of *MixedMutator* and the passed mutators. However, it is extremely important to understand that although *MixedMutator*(rates=mu) with *StepwiseMutator*(rates=1) and *MixedMutator*(rates=1) with *StepwiseMutator*(rates=mu) mutate alleles at the same mutation rate, the former is much more efficient because it triggers far less mutation events.

5.6.8 Context-dependent mutation models (`ContextMutator`)**

All mutation models we have seen till now are context independent. That is to say, how an allele is mutated depends only on the allele itself. However, it is understood that DNA and amino acid substitution rates are highly sequence context-dependent, e.g., C T substitutions in vertebrates may occur much more frequently at CpG sites. To simulate such models, a mutator must consider the context of a mutated allele, e.g. certain number of alleles to the left and right of this allele, and mutate the allele accordingly.

A `ContextMutator` can be used to mutate an allele depending on its surrounding loci. This mutator is constructed by a list of mutators and their respective contexts. It accepts regular mutator parameters such as `rates`, `loci`, `subPops`, `mapIn` and `mapOut` and mutates alleles at specified rate. When a mutation event happens, it checks the context of the mutated allele and choose a corresponding mutator to mutate the allele. An additional mutator can be specified to mutate alleles with unknown context. Example `ContextMutator` applies two `SNPMutator` at different rates under different contexts.

Example: *A context-dependent mutation model*

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=[3, 3])
>>> pop.evolve(
...     # initialize locus by 0, 0, 0, 1, 0, 1
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 1], loci=[3, 5])
...     ],
...     preOps=[
...         sim.ContextMutator(mutators=[
...             sim.SNPMutator(u=0.1),
...             sim.SNPMutator(u=1),
...         ],
...         contexts=[(0, 0), (1, 1)],
...         loci=[1, 4],
...         rates=0.01
...     ),
...         sim.Stat(alleleFreq=[1, 4], step=5),
...         sim.PyEval(r"Gen: %2d freq1: %.3f, freq2: %.3f\n" +
...             " % (gen, alleleFreq[1][1], alleleFreq[4][1])", step=5)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 20
... )
Gen:  0 freq1: 0.001, freq2: 0.010
Gen:  5 freq1: 0.005, freq2: 0.059
Gen: 10 freq1: 0.007, freq2: 0.108
Gen: 15 freq1: 0.015, freq2: 0.142
20

now exiting runScriptInteractively...
```

[Download ContextMutator.py](#)

Note that although

```
ContextMutator(mutators=[
    SNPMutator(u=0.1),
    SNPMutator(u=1)],
    contexts=[(0, 0), (1, 1)],
    rates=0.01
```

(continues on next page)

(continued from previous page)

)

and

```
ContextMutator(mutators=[
    SNPMutator(u=0.001),
    SNPMutator(u=0.01)],
    contexts=[(0, 0), (1, 1)],
    rates=1
)
```

both apply two *SNPMutator* at mutation rates 0.001 and 0.01, the former is more efficient because it triggers less mutation events.

Context-dependent mutator can also be implemented by a *PyMutator*. When a non-zero parameter `context` is specified, this mutator will collect `context` number of alleles to the left and right of a mutated allele and pass them as a second parameter of the user-provided mutation function. Example *pyContextMutator* applies the same mutation model as Example *ContextMutator* using a *PyMutator*.

Example: *A hybrid context-dependent mutation model*

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(5000, loci=[3, 3])
>>> def contextMut(allele, context):
...     if context == [0, 0]:
...         if allele == 0 and random.random() < 0.1:
...             return 1
...     elif context == [1, 1]:
...         if allele == 0:
...             return 1
...     # do not mutate
...     return allele
>>> pop.evolve(
...     # initialize locus by 0, 0, 0, 1, 0, 1
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[1, 1], loci=[3, 5])
...     ],
...     preOps=[
...         sim.PyMutator(func=contextMut, context=1,
...             loci=[1, 4], rates=0.01
...         ),
...         #sim.SNPMutator(u=0.01, v= 0.01, loci=[1, 4]),
...         sim.Stat(alleleFreq=[1, 4], step=5),
...         sim.PyEval(r"Gen: %2d freq1: %.3f, freq2: %.3f\n" +
...             " % (gen, alleleFreq[1][1], alleleFreq[4][1])", step=5)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 20
... )
Gen:  0 freq1: 0.000, freq2: 0.000
Gen:  5 freq1: 0.000, freq2: 0.000
Gen: 10 freq1: 0.000, freq2: 0.000
Gen: 15 freq1: 0.000, freq2: 0.000
20
```

(continues on next page)

```
now exiting runScriptInteractively...
```

[Download pyContextMutator.py](#)

5.6.9 Manually-introduced mutations (*PointMutator*)

Operator *PointMutator* is different from all other mutators in that it mutates specified alleles of specified individuals. It is usually used to manually introduce one or more mutants to a population. Although it is not a recommended method to introduce a disease predisposing allele, the following example (Example *PointMutator*) demonstrates an evolutionary process where mutants are repeatedly introduced and raised by positive selection until it reaches an appreciable allele frequency. This example uses two *IfElse* operators. The first one introduces a mutant when there is no mutant in the population, and the second one terminate the evolution when the frequency of the mutant reaches 0.05.

Example: *Use a point mutator to introduce a disease predisposing allele*

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1, infoFields='fitness')
>>> pop.evolve(
...     initOps=sim.PyOutput('Introducing alleles at generation'),
...     preOps=sim.MaSelector(loci=0, wildtype=0, fitness=[1, 1.05, 1.1]),
...     matingScheme=sim.RandomSelection(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.IfElse('alleleNum[0][1] == 0', ifOps=[
...             sim.PyEval(r"' %d' % gen"),
...             sim.PointMutator(inds=0, loci=0, allele=1),
...         ]),
...         sim.IfElse('alleleFreq[0][1] > 0.05', ifOps=[
...             sim.PyEval(r"'\nTerminate at generation %d at allele freq %.3f.\n'" +
...                 " % (gen, alleleFreq[0][1])"),
...             sim.TerminateIf('True'),
...         ])
...     ],
... )
Introducing alleles at generation 0 1 2 16 17 18 22 30 32 33 34 41 81 82 83.
Terminate at generation 111 at allele freq 0.051.
112

now exiting runScriptInteractively...
```

[Download PointMutator.py](#)

5.6.10 Apply mutation to (virtual) subpopulations *

A mutator is usually applied to all individuals in a population. However, you can restrict its use to specified subpopulations and/or virtual subpopulations using parameter *subPop*. For example, you can use *subPop*=[0, 2] to apply the mutator only to individuals in subpopulations 0 and 2.

Virtual subpopulations can also be specified in this parameter. For example, you can apply different mutation models to male and female individuals, to unaffected or affected individuals, to patients at different stages of a cancer. Example *mutatorVSP* demonstrate a mutation model where individuals with more tandem repeats at a disease predisposing

locus are more likely to develop a disease (e.g. fragile-X). Affected individuals are then subject to a non-neutral mutation model at an accelerated mutation rate.

Example: Applying mutation to virtual subpopulations.

```
>>> import simuPOP as sim
>>> def fragileX(geno):
...     '''A disease model where an individual has increased risk of
...     affected if the number of tandem repeats exceed 75.
...     '''
...     # Alleles A1, A2.
...     maxRep = max(geno)
...     if maxRep < 50:
...         return 0
...     else:
...         # individuals with allele >= 70 will surely be affected
...         return min(1, (maxRep - 50)*0.05)
...
>>> def avgAllele(pop):
...     'Get average allele by affection sim.status.'
...     sim.stat(pop, alleleFreq=(0,1), subPops=[(0,0), (0,1)],
...             numOfAffected=True, vars=['alleleNum', 'alleleNum_sp'])
...     avg = []
...     for alleleNum in [\
...         pop.dvars((0,0)).alleleNum[0], # first locus, unaffected
...         pop.dvars((0,1)).alleleNum[0], # first locus, affected
...         pop.dvars().alleleNum[1],      # second locus, overall
...     ]:
...         alleleSum = numAllele = 0
...         for idx,cnt in enumerate(alleleNum):
...             alleleSum += idx * cnt
...             numAllele += cnt
...         if numAllele == 0:
...             avg.append(0)
...         else:
...             avg.append(alleleSum * 1.0 / numAllele)
...     # unaffected, affected, loc2
...     pop.dvars().avgAllele = avg
...     return True
...
>>> pop = sim.Population(10000, loci=[1, 1])
>>> pop.setVirtualSplitter(sim.AffectionSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[50, 50])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # determine affection sim.status for each offspring (duringMating)
...         sim.PyPenetrance(func=fragileX, loci=0),
...         # unaffected offspring, mutation rate is high to save some time
...         sim.StepwiseMutator(rates=1e-3, loci=1),
...         # unaffected offspring, mutation rate is high to save some time
...         sim.StepwiseMutator(rates=1e-3, loci=0, subPops=[(0, 0)]),
...         # affected offspring have high probability of mutating upward
...         sim.StepwiseMutator(rates=1e-2, loci=0, subPops=[(0, 1)],
...             incProb=0.7, mutStep=3),
...     ],
... )
```

(continues on next page)

(continued from previous page)

```

...     # number of affected
...     sim.PyOperator(func=avgAllele, step=20),
...     sim.PyEval(r"Gen: %3d #Aff: %d AvgRepeat: %.2f (unaff), %.2f (aff), %.2f_
↪(unrelated)\n"
...         + " % (gen, numOfAffected, avgAllele[0], avgAllele[1], avgAllele[2])",
...         step=20),
...     ],
...     gen = 101
... )
Gen:  0 #Aff: 0 AvgRepeat: 1.01 (unaff), 0.00 (aff), 1.01 (unrelated)
Gen: 20 #Aff: 6 AvgRepeat: 1.53 (unaff), 0.50 (aff), 1.52 (unrelated)
Gen: 40 #Aff: 20 AvgRepeat: 2.56 (unaff), 2.04 (aff), 1.53 (unrelated)
Gen: 60 #Aff: 46 AvgRepeat: 2.56 (unaff), 2.04 (aff), 2.04 (unrelated)
Gen: 80 #Aff: 55 AvgRepeat: 3.08 (unaff), 1.53 (aff), 2.04 (unrelated)
Gen: 100 #Aff: 48 AvgRepeat: 2.04 (unaff), 1.52 (aff), 2.04 (unrelated)
101

now exiting runScriptInteractively...
```

Download mutatorVSP.py

At the beginning of a simulation, all individuals have 50 copies of a tandem repeat and the mutation follows a standard neutral stepwise mutation model. individuals with more than 50 repeats will have an increasing probability to develop a disease () for). The average repeat number therefore increases for affected individuals. In contrast, the mean number of repeats at locus 1 on a separate chromosome oscillate around 50.

5.6.11 Allele mapping **

If alleles in your simulation do not follow the convention of a mutation model, you may want to use the `pop.recodeAlleles()` function to recode your alleles so that appropriate mutation models could be applied. If this is not possible, you can use a general mutation model with your own mutation matrix, or an advanced feature called **allele mapping**.

Allele mapping is done through two parameters *mapIn* and *mapOut*, which map alleles in your population to and from alleles assumed in a mutation model. For example, an *AcgtMutator* mutator assumes alleles A, C, G and T for alleles 0, 1, 2, and 3 respectively. If for any reason the alleles in your application does not follow this order, you will need to map these alleles to the alleles assumed in the mutator. For example, if you assumes C, G, A, T for alleles 0, 1, 2, and 3 respectively, you can use parameters

```
mapIn=[1, 2, 0, 3], mapOut=[2, 0, 1, 3]
```

to map your alleles (C (0) → C (1), G (1) → G (2), A (2) → A (0), T (3) → T (3)) to alleles *AcgtMutator* assumes, and then map mutated alleles (A (0) → A (2), C (1) → C (0), G (2) → G (1), T (3) → T (3)) back. Example *alleleMapping* gives another example where alleles 4, 5, 6 and 7 are mutated using a 4-allele model.

Example: Allele mapping for mutation operators

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[2000], loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0]*4 + [0.1, 0.2, 0.3, 0.4])
...     ],
...     matingScheme=sim.RandomMating(),
```

(continues on next page)

(continued from previous page)

```

...     postOps=[
...         sim.KAlleleMutator(k=4, rates=1e-4, mapIn=[0]*4 + list(range(4)),
...             mapOut=[4, 5, 6, 7]),
...         sim.Stat(alleleFreq=0, step=100),
...         sim.PyEval(r"', '.join(['%.2f' % alleleFreq[0][x] for x in range(8)]) +
...             ↪ '\n"',
...             step=100),
...     ],
...     gen=500
... )
0.00, 0.00, 0.00, 0.00, 0.09, 0.20, 0.30, 0.41
0.00, 0.00, 0.00, 0.00, 0.13, 0.20, 0.40, 0.26
0.00, 0.00, 0.00, 0.00, 0.17, 0.20, 0.31, 0.31
0.00, 0.00, 0.00, 0.00, 0.19, 0.18, 0.26, 0.37
0.00, 0.00, 0.00, 0.00, 0.18, 0.24, 0.23, 0.34
500

now exiting runScriptInteractively...

```

Download alleleMapping.py

These two parameters also accept Python functions which should return corresponding mapped-in or out allele for a given allele. These two functions can be used to explore very fancy mutation models. For example, you can categorize a large number of alleles into alleles assumed in a mutation model, and emit random alleles from a mutated allele.

5.6.12 Mutation rate and transition matrix of a *MatrixMutator***

A *MatrixMutator* is specified by a mutation rate matrix. Although mutation rates of this mutator is typically allele-dependent, the *MatrixMutator* is implemented as a two-step process where mutation events are triggered independent to allelic states. This section describes these two steps which can be useful if you need to use a *MatrixMutator* in a *MixedMutator* or *ContextMutator*, and would like to factor out an allele-independent mutation rate to the wrapper mutator.

Because alleles usually have different probabilities of mutating to other alleles, **a mutation process is usually allele dependent**. Given a mutation model, it is obviously inefficient to go through all mutable alleles and determine whether or not to mutate it using `.simuPOP` uses a two step procedure to mutate a large number of alleles. More specifically, for each mutation model, we determine as the overall mutation rate, and then

1. For each allele, trigger a mutation event with probability `.`. Because `.` is usually very small and is the same for all alleles, this step can be implemented efficiently.
2. When a mutation event happens, mutation allele to allele with probability

Because steps 1 and 2 are independent, it is easy to verify that

if and

where the first and second items are probabilities of no-mutation at steps 1 and 2. `.` was chosen as the smallest that makes for all `.`

For example, for a `-`allele model with

is directly for the first step and

for the second step. Therefore, mutation rate in a `-`allele model could be interpreted as the probability of mutation, and a mutation event would mutate an allele to any other allele with equal probability.

For a classical mutation model with and ,

if and , ,

That is to say, we would mutate at a mutation rate , mutate allele to with probability 1 and mutate allele to with probability 0.5.

5.6.13 Infinite-sites model and other simulation techniques **

Infinite-sites and infinite-alleles models have some similarities. If you assume that mutation is the only force to create new mutants, you can treat a long chromosomal region as a locus and use the infinite-alleles model, actually a -allele model with large , to mimic the infinite-site model. This assumption is certainly wrong with the infinite-site model when recombination is involved, because recombination creates new haplotypes (alleles) under the infinite-site model. However, for short regions where recombination can be ignored, an -allele model can be an easy and fast way to mimic an infinite-site model. That statement basically says that you have a choice between two models if you would like to simulate the evolution of this gene, namely considering the gene as a locus and simulating variants as alleles, or considering the gene as a sequence and simulating haplotypes as alleles.

For example, the CFTR gene (for cystic fibrosis) can have many alleles (thinking in terms of infinite-allele model) which are nucleotide mutations on tens of locations (infinite-site model). In order to simulate the evolution of this gene, you have a choice between two models, namely considering the gene as a locus and simulating variants as alleles, or considering the gene as a sequence and simulating haplotypes as alleles. Because there is supposed to be only one mutant at each site, you can assign a unique *location* for each allele of an infinite-allele model and convert multi-allelic datasets simulated by an infinite-allele model to sequences of diallelic markers. Note that mutation rates are interpreted differently for these two models.

If specific location of such a mutation is needed, it is possible to record the location of mutations during an evolution and mimic an infinite-sites model. For example, alleles in Example *infiniteSites* are used to store location of a mutation event. When a mutation event happens, the location of the new allele (rather the allele itself) is recorded on the chromosome (actually list of mutation events) of an individual. The transmission of chromosomes proceed normally and effectively transmit mutants from parents to offspring. At the end of the simulation, each individual accumulates a number of mutation events and they are essentially alleles at their respective locations.

Example: *Mimicking an infinite-sites model using mutation events as alleles*

```
>>> import simuOpt
>>> simuOpt.setOptions(alleleType='long')
>>> import simuPOP as sim
>>>
>>> def infSitesMutate(pop, param):
...     '''Apply an infinite mutation model'''
...     (startPos, endPos, rate) = param
...     # for each individual
...     for ind in pop.individuals():
...         # for each homologous copy of chromosomes
...         for p in range(2):
...             # using a geometric distribution to determine
...             # the first mutation location
...             loc = sim.getRNG().randGeometric(rate)
...             # if a mutation happens, record the mutated location
...             if startPos + loc < endPos:
...                 try:
...                     # find the first non-zero location
...                     idx = ind.genotype(p).index(0)
...                     # record mutation here
...                     ind.setAllele(startPos + loc, idx, ploidy=p)
...                 except:
...                     raise
```

(continues on next page)

(continued from previous page)

```

...         print('Warning: more than %d mutations have accumulated' %
↳pop.totNumLoci())
...         pass
...     return True
...
>>> pop = sim.Population(size=[2000], loci=[100])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         # mutate in a 10Mb region at rate 1e-8
...         sim.PyOperator(func=infSitesMutate, param=(1, 10000000, 1e-8)),
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 100
... )
100
>>> # now, we get a sim.Population. Let us have a look at the 'alleles'.
>>> # print the first five mutation locations
>>> print(pop.individual(0).genotype()[5])
[1527502, 4774892, 7979220, 3671118, 395142]
>>> # how many alleles are there (does not count 0)?
>>> print(len(set(pop.genotype())) - 1)
2700
>>> # Allele count a simple count of alleles.
>>> cnt = {}
>>> for allele in pop.genotype():
...     if allele == 0:
...         continue
...     if allele in cnt:
...         cnt[allele] += 1
...     else:
...         cnt[allele] = 1
...
>>> # highest allele frequency?
>>> print(max(cnt.values()) * 0.5 / pop.popSize())
0.05475

now exiting runScriptInteractively...
```

Download infiniteSites.py

All mutation models in simuPOP apply to existing alleles at pre-specified loci. However, if the location of loci cannot be determined beforehand, it is sometimes desired to create new loci as a result of mutation. A customized operator can be used for this purpose (see Example *newOperator*), but extra attention is needed to make sure that other operators are applied to the correct loci because loci indexes will be changed with the insertion of new loci. This technique could also be used to simulate mutations over long sequences.

5.6.14 Recording and tracing individual mutants **

Mutation operators mutate alleles in place and by default do not generate any output. If you are interested in knowing the source of each mutant, you can specify an output stream and let the mutation operators dump details of each mutation event, which consists of generation number, locus index, ploidy, original allele, and mutated allele. If a list of information fields are specified through parameter `infoFields`, values at these information fields will also be outputted (if they exist in the population). The default information field is `ind_id`, which allow you to record the ID of individuals harboring the mutants.

Example `countMutants` demonstrates how to use this feature to count the number of mutants at each locus. Instead of sending the output to a file (e.g. `output='>>mutants.txt'`), this example sends the output to a Python function, which parses input string and counts the number of mutants at each locus using a global dictionary variable. As we can see from the output, because the `KAlleleMutator` uses a higher mutation rate (0.01) at locus 1 than mutation rate (0.001) at locus 0, there are 10 times more mutants at the second locus. There are about 3/4 mutations on the locus on chromosome X and 1/4 mutations on the locus on chromosome Y, for obvious reasons.

Example: *Count number of mutants from mutator outputs*

```
>>> import simuPOP as sim
>>> from collections import defaultdict
>>> # count number of mutants at each locus
>>> counter = defaultdict(int)
>>> def countMutants(mutants):
...     global counter
...     for line in mutants.split('\n'):
...         # a trailing \n will lead to an empty string
...         if not line:
...             continue
...         (gen, loc, ploidy, a1, a2, id) = line.split('\t')
...         counter[int(loc)] += 1
...
>>> pop = sim.Population([5000]*3, loci=[2,1,1], infoFields='ind_id',
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.IdTagger(),
...     ],
...     preOps=[
...         sim.KAlleleMutator(rates=[0.001] + [0.01]*3,
...             loci=range(4), k=100, output=countMutants),
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.IdTagger(),
...             sim.MendelianGenoTransmitter()
...         ],
...         gen = 10
...     )
... )
10
>>> print(counter.items())
dict_items([(0, 308), (1, 2984), (2, 2319), (3, 768)])

now exiting runScriptInteractively...
```

[Download countMutants.py](#)

5.7 Penetrance

Penetrance is the probability for an individual to be affected with a disease conditioning on his or her genotype and other risk factors. A penetrance model calculates such a probability for an individual and assign affection status randomly according to this probability. For example, if an individual with genotype 10 has probability 0.2 to be affected according to a penetrance model, he or she will be affected with probability 0.2. Note that simuPOP supports only one affection status. If there are multiple affection outcomes involved, you can treat them as binary quantitative

traits and use information fields to store them.

A penetrance operator can be applied before or after mating, to assign affection status to all individuals in the parental or offspring generation, respectively. It can also be applied during mating and assign affection status to each offspring. The latter could be used to assist natural selection through the selection of offspring. You can also assign affection status to all individuals in a population using the function form of a penetrance operator (e.g. function `mapPenetrance` for operator *MapPenetrance*). Compared the penetrance operators that assign affection status to only the current generation, **these functions by default assign affection status to all ancestral generations as well.**

A penetrance operator usually do not store the penetrance values. However, if an information field is given, penetrance values will be saved to this information field before it is used to determine individual affection status.

5.7.1 Map penetrance model (operator *MapPenetrance*)

A map penetrance operator uses a Python dictionary to provide penetrance values for each type of genotype. For example, Example *MapPenetrance* uses a dictionary with keys $(0,0)$, $(0,1)$ and $(1,1)$ to specify penetrance for individuals with these genotypes at locus 0.

Example: *A penetrance model that uses pre-defined fitness value*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=2)
>>> sim.initGenotype(pop, freq=[.2, .8])
>>> sim.mapPenetrance(pop, loci=0,
...     penetrance={(0,0):0, (0,1):.2, (1,1):.3})
>>> sim.stat(pop, genoFreq=0, numOfAffected=1, vars='genoNum')
>>> # number of affected individuals
>>> pop.dvars().numOfAffected
531
>>> # which should be roughly (#01 + #10) * 0.2 + #11 * 0.3
>>> (pop.dvars().genoNum[0][(0,1)] + pop.dvars().genoNum[0][(1,0)]) * 0.2 \
... + pop.dvars().genoNum[0][(1,1)] * 0.3
514.2

now exiting runScriptInteractively...
```

[Download MapPenetrance.py](#)

The above example assumes that penetrance for individuals with genotypes $(0,1)$ and $(1,0)$ are the same. This assumption is usually valid but can be violated with imprinting. In that case, you can specify fitness for both types of genotypes. The underlying mechanism is that the *MapPenetrance* looks up a genotype in the dictionary first directly, and then without phase information if a genotype is not found.

This operator supports haplodiploid populations and sex chromosomes. In these cases, only valid alleles should be listed which can lead to dictionary keys with different lengths. In addition, although less used because of potentially a large number of keys, this operator can act on multiple loci. For example,

- keys $(a1,a2)$ and $(a1,)$ can be used to specify fitness values for female and male individuals in a haplodiploid population, respectively
- keys $(x1,x2)$ and $(x1,)$ can be used to specify fitness for female and male individuals according to a locus on the X chromosome in a diploid population, respectively. Similarly, keys $()$ and $(y,)$ for a locus on chromosome Y.
- keys $(a1,a2,b1,b2)$ can be used to specify fitness values according to genotype at two loci in a diploid population.

5.7.2 Multi-allele penetrance model (operator `MaPenetrance`)

A multi-allele penetrance model divides alleles into two groups, wildtype A and mutants a , and treat alleles within each group as the same. The penetrance model is therefore simplified to

- Two fitness values for genotype , in the haploid case
- Three fitness values for genotype AA , Aa and aa in the diploid single locus case. Genotype Aa and aA are assumed to have the same impact on fitness.

The default wildtype group contains allele 0 so the two allele groups are zero and non-zero alleles. Example [MaPenetrance](#) demonstrates the use of this operator.

Example: *A multi-allele penetrance model*

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=3)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.9] + [0.02]*5)
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.MaPenetrance(loci=0, penetrance=(0.01, 0.2, 0.3)),
...         sim.Stat(numOfAffected=True, vars='propOfAffected'),
...         sim.PyEval(r"Gen: %d Prevalence: %.1f%%\n" % (gen, propOfAffected*100)),
...     ],
...     gen = 5
... )
Gen: 0 Prevalence: 4.4%
Gen: 1 Prevalence: 4.4%
Gen: 2 Prevalence: 4.7%
Gen: 3 Prevalence: 4.4%
Gen: 4 Prevalence: 4.3%
5
now exiting runScriptInteractively...
```

[Download MaPenetrance.py](#)

Operator [MaPenetrance](#) also supports multiple loci by specifying fitness values for all combination of genotype at specified loci. In the case of two loci, this operator requires

- Four fitness values for genotype AB , Ab , aB and ab in the haploid case,
- Nine fitness values for genotype $AABB$, $AABb$, $AAbb$, $AaBB$, $AaBb$, $Aabb$, $aaBB$, $aaBb$, and $aabb$ in the haploid case.

In general, values are needed for haploid populations and values are needed for diploid populations where is the number of loci. This operator does not yet support haplodiploid populations and sex chromosomes.

5.7.3 Multi-loci penetrance model (operator `MlPenetrance`)

Although an individual's affection status can be affected by several factors, each of which can be modeled individually, **only one penetrance value is used to determine a person's affection status** and we have to use a multi-locus penetrance model to combine single-locus models.

This multi-loci penetrance model applies several penetrance models to each Individual and computes an overall penetrance value from the penetrance values provided by these operators. Although this selector is designed to obtain

multi-loci penetrance values from several single-locus penetrance models, any penetrance operator, including those obtain their penetrance values from multiple disease predisposing loci, can be used in this operator. This operator uses parameter `mode` to control how Individual penetrance values are combined. More specifically, if are penetrance values obtained from individual selectors, this selector returns

- if `mode=MULTIPLICATIVE`, and
- if `mode=ADDITIVE`, and
- if `mode=HETEROGENEITY`

0 or 1 will be returned if the returned fitness value is out of range of $[0, 1]$.

Example *MLPenetrance* demonstrates the use of this operator using an multiplicative multi-locus model over three additive single-locus models at three disease predisposing loci.

Example: A multi-loci penetrance model

```
>>> import simuPOP as sim
>>> pop = sim.Population(5000, loci=3)
>>> sim.initGenotype(pop, freq=[0.2]*5)
>>> # the multi-loci penetrance
>>> sim.mLPenetrance(pop, mode=sim.MULTIPLICATIVE,
...     ops = [sim.MaPenetrance(loci=loc,
...     penetrance=[0, 0.3, 0.6]) for loc in range(3)])
>>> # count the number of affected individuals.
>>> sim.stat(pop, numOfAffected=True)
>>> pop.dvars().numOfAffected
542

now exiting runScriptInteractively...
```

[Download MLPenetrance.py](#)

5.7.4 Hybrid penetrance model (operator *PyPenetrance*)

When your selection model involves multiple interacting genetic and environmental factors, it might be easier to calculate a penetrance value explicitly using a Python function. A hybrid penetrance operator can be used for this purpose. If your penetrance model depends solely on genotype, you can define a function such as

```
def pfunc(geno):
    # calculate penetrance according to genotype at specified loci
    # in the order of A1,A2,B1,B2,C1,C2 for loci A,B,C (for diploid)
    return val
```

and use this function in an operator *PySelector*(`func=pfunc, loci=loci`). If your penetrance model depends on genotype as well as some information fields, you can define a function in the form of

```
def pfunc(geno, fields):
    # calculate penetrance according to genotype at specified loci
    # and values at specified informaton fields.
    return val
```

and use this function in an operator *PySelector*(`func=pfunc, loci=loci, paramFields=fields`). If the function you provide accepts three arguments, *PyPenetrance* will pass generation number as the third argument so that you could implement generation-specific penetrance models (e.g. `pfunc(geno, fields, gen)`).

When a *PyPenetrance* operator is used to calculate penetrance for an individual, it will collect his or her genotype at specified loci, optional values at specified information fields, and the generation number to a user- specified Python

function, and take its return value as penetrance. As you can imagine, the incorporation of information fields and generation number allow the implementation of very complex penetrance scenarios such as gene environment interaction and varying selection pressures. Note that this operator does not pass sex and affection status to the user-defined function. If your selection model is sex-dependent, you can define an information field `sex`, synchronize its value with individual sex (e.g. using operator `InfoExec('sex=ind.sex()', exposeInd='ind')`) and pass this information to the user-defined function (`PySelector(func=func, paramFields='sex')`).

Example `PySelector` demonstrates how to use a `PyPenetrance` to specify penetrance values according to a fitness table and the smoking status of each individual. In this example, Individual risk is doubled when he or she smokes. The disease prevalence is therefore much higher in smokers than in non-smokers.

Example: A hybrid penetrance model

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=2000, loci=[1]*2, infoFields=['p', 'smoking'])
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='smoking', values=[0,1]))
>>> # the second parameter gen can be used for varying selection pressure
>>> def penet(gen, smoking):
...     #      BB      Bb      bb
...     # AA  0.01   0.01   0.01
...     # Aa  0.01   0.03   0.03
...     # aa  0.01   0.03   0.05
...     #
...     # geno is (A1 A2 B1 B2)
...     if geno[0] + geno[1] == 1 and geno[2] + geno[3] != 0:
...         v = 0.03 # case of AaBb
...     elif geno[0] + geno[1] == 2 and geno[2] + geno[3] == 1:
...         v = 0.03 # case of aaBb
...     elif geno[0] + geno[1] == 2 and geno[2] + geno[3] == 2:
...         v = 0.05 # case of aabb
...     else:
...         v = 0.01 # other cases
...     if smoking:
...         return v * 2
...     else:
...         return v
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5]),
...         sim.PyOutput('Calculate prevalence in smoker and non-smokers\n'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # set smoking status randomly
...         sim.InitInfo(lambda : random.randint(0,1), infoFields='smoking'),
...         # assign affection status
...         sim.PyPenetrance(loci=[0, 1], func=penet),
...         sim.Stat(numOfAffected=True, subPops=[(0, sim.ALL_AVAIL)],
...             vars='propOfAffected_sp', step=20),
...         sim.PyEval(r"Non-smoker: %.2f%%\tSmoker: %.2f%%\n" % "
...             "(subPop[(0,0)][ 'propOfAffected' ]*100, subPop[(0,1)][ 'propOfAffected
...             ↪']*100)",
...             step=20)
...     ],
...     gen = 50
```

(continues on next page)

(continued from previous page)

```

... )
Calculate prevalence in smoker and non-smokers
Non-smoker: 2.24%      Smoker: 4.52%
Non-smoker: 2.29%      Smoker: 3.61%
Non-smoker: 1.85%      Smoker: 3.80%
50
>>>

now exiting runScriptInteractively...

```

[Download PyPenetrance.py](#)

5.8 Quantitative trait

Quantitative traits are naturally stored in information fields of each individual. A quantitative trait operator assigns quantitative trait fields according to individual genetic (genotype) and environmental (other information fields) information. Although a large number of quantitative trait models have been used in theoretical and empirical studies, no model is popular enough to deserve a specialized operator. Therefore, only one hybrid operator is currently provided in simuPOP.

5.8.1 A hybrid quantitative trait operator (operator `PyQuanTrait`)

Operator `PyQuanTrait` accepts a user defined function that returns quantitative trait values for specified information fields. This operator can communicate with functions in one of the forms of `func(geno)`, `func(geno, field_name, ...)` or `func(geno, field_name, gen)` where `field_name` should be name of existing fields. simuPOP will pass genotype and value of specified fields according to name of the passed function. Note that `geno` are arranged locus by locus, namely in the order of A1, 'A2', 'B1', 'B2' for loci A and B.

A quantitative trait operator can be applied before or after mating and assign values to the trait fields of all parents or offspring, respectively. It can also be applied during mating to assign trait values to offspring. Example `PyQuanTrait` demonstrates the use of this operator, using two trait fields `trait1` and `trait2` which are determined by individual genotype and age. This example also demonstrates how to calculate statistics within virtual subpopulations (defined by age).

Example: *A hybrid quantitative trait model*

```

>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=5000, loci=2, infoFields=['qtrait1', 'qtrait2', 'age'])
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', cutoff=[40]))
>>> def qtrait(geno, age):
...     'Return two traits that depends on genotype and age'
...     return random.normalvariate(age * sum(geno), 10), random.randint(0,
↳ 10*sum(geno))
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # use random age for simplicity

```

(continues on next page)

(continued from previous page)

```

...     sim.InitInfo(lambda:random.randint(20, 75), infoFields='age'),
...     sim.PyQuanTrait(loci=(0,1), func=qtrait, infoFields=['qtrait1', 'qtrait2
→']),
...     sim.Stat(meanOfInfo=['qtrait1'], subPops=[(0, sim.ALL_AVAIL)],
...             vars='meanOfInfo_sp'),
...     sim.PyEval(r"Mean of trait1: %.3f (age < 40), %.3f (age >=40)\n" % "
...             "(subPop[(0,0)]['meanOfInfo']['qtrait1'], subPop[(0,1)]['meanOfInfo']["
→'qtrait1'])"),
...     ],
...     gen = 5
... )
Mean of trait1: 92.876 (age < 40), 183.515 (age >=40)
Mean of trait1: 94.041 (age < 40), 183.374 (age >=40)
Mean of trait1: 95.447 (age < 40), 183.288 (age >=40)
Mean of trait1: 95.017 (age < 40), 183.919 (age >=40)
Mean of trait1: 94.769 (age < 40), 185.430 (age >=40)
5
>>>

now exiting runScriptInteractively...

```

[Download PyQuanTrait.py](#)

5.9 Natural Selection

5.9.1 Natural selection through the selection of parents

In the simplest scenario, natural selection is implemented in two steps:

- Before mating happens, an operator (called a **selector**) goes through a population and assign each individual a fitness value. The fitness values are stored in an information field called `fitness`.
- When mating happens, parents are chosen with probabilities that are proportional to their fitness values. For example, assuming that a parental population consists of four Individuals with fitness values 1, 2, 3, and 4, respectively, the probability that they are picked to produce offspring are $\frac{1}{10}$, $\frac{2}{10}$, $\frac{3}{10}$, and $\frac{4}{10}$ respectively. As you can image, if the offspring population has 10 individuals, the four parents will on average parent 1, 2, 3 and 4 offspring.

Because parents with lower fitness values have less chance to be produce offspring, their genotypes have less chance to be passed to an offspring generation. If the decreased fitness is caused by the presence of certain mutant (e.g. a mutant causing a serious disease), individuals with that mutant will have less change to survive and effectivly reduce or eliminate that mutant from the population.

Example `selectParents` gives an example of natural selection. In this example, a `MapSelector` is used to explicitly assign fitness value to genotypes at the first locus. The fitness values are 1, 0.98, 0.97 for genotypes 00, 01 and 11 respectively. The selector set individual fitness values to information field `fitness` before mating happens. The `RandomMating` mating scheme then selects parents according to parental fitness values.

Example: *Natural selection through the selection of parents*

```

>>> import simuPOP as sim
>>> pop = sim.Population(4000, loci=1, infoFields='fitness')
>>> simu = sim.Simulator(pop, rep=3)
>>> simu.evolve(
...     initOps=[

```

(continues on next page)

(continued from previous page)

```

...     sim.InitSex(),
...     sim.InitGenotype(freq=[0.5, 0.5])
... ],
... preOps=sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):0.98, (1,1):0.97}),
... matingScheme=sim.RandomMating(),
... postOps=[
...     sim.Stat(alleleFreq=0, step=10),
...     sim.PyEval("'Gen:%3d ' % gen", reps=0, step=10),
...     sim.PyEval(r"'.3f\t' % alleleFreq[0][1]", step=10),
...     sim.PyOutput('\n', reps=-1, step=10)
... ],
...     gen = 50
... )
Gen:  0 0.490      0.492   0.487
Gen: 10 0.433      0.430   0.431
Gen: 20 0.403      0.390   0.419
Gen: 30 0.343      0.325   0.383
Gen: 40 0.303      0.297   0.334
(50, 50, 50)

now exiting runScriptInteractively...
```

[Download selectParents.py](#)

Note: The selection algorithm used in simuPOP is called *fitness proportionate selection*, or *roulette-wheel selection*. simuPOP does not use the more efficient *stochastic universal sampling* algorithm because the number of needed offspring is unknown in advance.

5.9.2 Natural selection through the selection of offspring *

Natural selection can also be implemented as selection of offspring. Remember that an individual will be discarded if one of the during-mating operators fails (return `False`), **a during-mating selector discards offspring according to fitness values of offspring**. Instead of relative fitness that will be compared against other individuals during the selection of parents, **fitness values of a during-mating selector are considered as absolute fitness which are probabilities to survive** and have to be between 0 and 1.

A during-mating selector works as follows:

1. During evolution, parents are chosen randomly to produce one or more offspring. (Nothing prevents you from choosing parents according to their fitness values, but it is rarely justifiable to apply natural selection to both parents and offspring.)
2. A selection operator is applied to each offspring during mating and determines his or her fitness value. The fitness value is considered as probability to survive so an offspring will be discarded (operator returns `False`) if the fitnessvalue is larger than a uniform random number.
3. Repeat steps 1 and 2 until the offspring generation is populated.

Because many offspring will be generated and discarded, especially when offspring fitness values are low, selection through offspring is less efficient than selection through parents. In addition, absolute fitness is usually more difficult to estimate than relative fitness. So, unless there are compelling reasons (e.g. simulating realistic scenarios of survival competition among offspring), selection through parents are recommended.

Example [selectOffspring](#) gives an example of natural selection through the selection of offspring. This example looks almost identical to Example [selectParents](#) but the underlying selection mechanism is quite different. Note that selection

through offspring does not save fitness values to an information field so you do not need to add information field fitness to the population.

Example: *Natural selection through the selection of offspring*

```
>>> import simuPOP as sim
>>> pop = sim.Population(10000, loci=1)
>>> simu = sim.Simulator(pop, rep=3)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):0.98, (1,1):0.97}),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.PyEval("'Gen:%3d ' % gen", reps=0, step=10),
...         sim.PyEval(r"%.3f\t' % alleleFreq[0][1]", step=10),
...         sim.PyOutput('\n', reps=-1, step=10)
...     ],
...     gen = 50
... )
Gen:  0 0.493      0.493    0.496
Gen: 10 0.461      0.464    0.465
Gen: 20 0.436      0.445    0.442
Gen: 30 0.389      0.386    0.385
Gen: 40 0.370      0.345    0.348
(50, 50, 50)

now exiting runScriptInteractively...
```

[Download selectOffspring.py](#)

5.9.3 Are two selection scenarios equivalent? **

If you look closely at Examples *selectParents* and *selectOffspring*, you will notice that their results are quite similar. This is actually what you should expect in most cases. Let us look at the theoretical consequence of selection through parents or offspring in a simple case with asexual mating.

Assuming a diallelic marker with three genotypes, and, with frequencies, and, and relative fitness values, , , and respectively. If we select through offspring, the proportion of genotype etc., should be

because offspring genotypes are randomly drawn from the parental generation, and each offspring has certain probability to survive.

Now, if we select through parents, the proportion of parents with genotype will be the number of individuals times its probability to be chosen:

This is, however, exactly

which corresponds to the proportion of offspring with such genotype. That is to say, **in this simple case, two types of selection scenarios yield identical results.**

These two types of selection scenarios do not have to always yield identical results. Exceptions exist in cases with more than one offspring or sexual mating with sex-specific survival rate. simuPOP provides both selection implementations and you should choose one of them for your particular simulation.

5.9.4 Map selector (operator MapSelector)

A map selector uses a Python dictionary to provide fitness values for each type of genotype. For example, Example [MapSelector](#) uses a dictionary with keys (0,0), (0,1) and (1,1) to specify fitness values for individuals with these genotypes at locus 0. This example is a typical example of heterozygote advantage. When the genotype frequencies will go to an equilibrium state. Theoretically, if and , the stable allele frequency of allele 0 is

which is in the example (,).

Example: A selector that uses pre-defined fitness value

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=1, infoFields='fitness')
>>> s1 = .1
>>> s2 = .2
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.2, .8])
...     ],
...     preOps=sim.MapSelector(loci=0, fitness={(0,0):1-s1, (0,1):1, (1,1):1-s2}),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r"'.4f\n' % alleleFreq[0][0]", step=100)
...     ],
...     gen=301
... )
0.2250
0.6605
0.6530
0.6870
301
>>>

now exiting runScriptInteractively...
```

[Download MapSelector.py](#)

The above example assumes that the fitness value for individuals with genotypes (0,1) and (1,0) are the same. This assumption is usually valid but can be violated with imprinting. In that case, you can specify fitness for both types of genotypes. The underlying mechanism is that the [MapSelector](#) looks up a genotype in the dictionary first directly, and then without phase information if a genotype is not found.

This operator supports haplodiploid populations and sex chromosomes. In these cases, only valid alleles should be listed which can lead to dictionary keys with different lengths. In addition, although less used because of potentially a large number of keys, this operator can act on multiple loci. Please refer to [MapPenetrance](#) for details.

5.9.5 Multi-allele selector (operator MaSelector)

A multi-allele selector divides alleles into two groups, wildtype *A* and mutants *a*, and treat alleles within each group as the same. The fitness model is therefore simplified to

- Two fitness values for genotype , in the haploid case
- Three fitness values for genotype *AA*, *Aa* and *aa* in the diploid single locus case. Genotype *Aa* and *aA* are assumed to have the same impact on fitness.

The default wildtype group contains allele 0 so the two allele groups are zero and non-zero alleles. Example *MaSelector* demonstrates the use of this operator. This example is identical to Example *MapSelector* except that there are five alleles at locus 0 and alleles 1, 2, 3, 4 are treated as a single non-wildtype group.

Example: A multi-allele selector

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=1, infoFields='fitness')
>>> s1 = .1
>>> s2 = .2
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.2] * 5)
...     ],
...     preOps=sim.MaSelector(loci=0, fitness=[1-s1, 1, 1-s2]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r"%0.4f\n" % alleleFreq[0][0], step=100)
...     ],
...     gen = 301)
0.2250
0.6605
0.6530
0.6870
301

now exiting runScriptInteractively...
```

Download *MaSelector.py*

Operator *MaSelector* also supports multiple loci by specifying fitness values for all combination of genotype at specified loci. In the case of two loci, this operator requires

- Four fitness values for genotype AB, Ab, aB and ab in the haploid case,
- Nine fitness values for genotype AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, and aabb in the haploid case.

In general, values are needed for haploid populations and values are needed for diploid populations where is the number of loci. This operator does not yet support haplodiploid populations and sex chromosomes. Example *MaSelectorHaploid* demonstrates the use of a multi-locus model in a haploid population.

Example: A multi-locus multi-allele selection model in a haploid population

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, ploidy=1, loci=[1,1], infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     # fitness values for AB, Ab, aB and ab
...     preOps=sim.MaSelector(loci=[0,1], fitness=[1, 1, 1, 0.95]),
...     matingScheme=sim.RandomSelection(),
...     postOps=[
...         sim.Stat(haploFreq=[0, 1], step=25),
...         sim.PyEval(r"%0.3f\t%0.3f\t%0.3f\t%0.3f\n" % (haploFreq[(0,1)][(0,0)],
...             "haploFreq[(0,1)][(0,1)], haploFreq[(0,1)][(1,0)],"
```

(continues on next page)

(continued from previous page)

```

...             "haploFreq[(0,1)][(1,1)]", step=25)
...         ],
...         gen = 100
...     )
0.264         0.243    0.252    0.240
0.292         0.294    0.321    0.093
0.339         0.330    0.303    0.027
0.310         0.383    0.297    0.009
100
now exiting runScriptInteractively...

```

[Download MaSelectorHaploid.py](#)

5.9.6 Multi-locus selection models (operator `MlSelector`)

Although an individual's fitness can be affected by several factors, each of which can be modeled individually, **only one fitness value is used to determine a person's ability to pass all these factors to his or her offspring**. Although in theory we sometimes assume independent evolution of disease predisposing loci (mostly for mathematical reasons), in practise we have to use a multi-locus selection model to combine single-locus models.

This multi-loci selector applies several selectors to each individual and computes an overall fitness value from the fitness values provided by these selectors. Although this selector is designed to obtain multi-loci fitness values from several single-locus fitness models, any selector, including those obtain their fitness values from multiple disease predisposing loci, can be used in this selector. This selector uses parameter `mode` to control how individual fitness values are combined. More specifically, if are fitness values obtained from individual selectors, this selector returns

- if `mode=MULTIPLICATIVE`, and
- if `mode=ADDITIVE`, and
- if `mode=HETEROGENEITY`

0 will be returned if the returned fitness value is less than 0.

This operator simply combines individual fitness values and it is your responsibility to apply and interpret these models. For example, if relative fitness values are greater than one, the heterogeneity model hardly makes sense. Example *MlSelector* demonstrates the use of this operator using an additive multi-locus model over an additive and a recessive single- locus model at two disease predisposing loci. For comparison, we simulate two additional replicates with selection only applying to one of the two loci. It would be interesting to see if these two loci evolve more or less independently by comparing allele frequency trajectories of these two replicates to those in the first replicate.

Example: *A multi-loci selector*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, loci=2, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=[
...         sim.MlSelector([
...             sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):1, (1,1):.8}),
...             sim.MapSelector(loci=1, fitness={(0,0):1, (0,1):0.9, (1,1):.8}),
...             ], mode = sim.ADDITIVE, reps=0),
...         sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):1, (1,1):.8}, reps=1),
...     ]

```

(continues on next page)

(continued from previous page)

```

...     sim.MapSelector(loci=1, fitness={(0,0):1, (0,1):0.9, (1,1):.8}, reps=2)
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=[0,1]),
...         sim.PyEval(r"'REP %d:\t%.3f\t%.3f\t' % (rep, alleleFreq[0][1],
↪alleleFreq[1][1])"),
...         sim.PyOutput('\n', reps=-1),
...     ],
...     gen = 5
... )
REP 0:      0.472    0.465
REP 0:      0.452    0.429
REP 0:      0.429    0.397
REP 0:      0.405    0.378
REP 0:      0.382    0.355
5

now exiting runScriptInteractively...
```

[Download MlSelector.py](#)

5.9.7 A hybrid selector (operator `PySelector`)

When your selection model involves multiple interacting genetic and environmental factors, it might be easier to calculate a fitness value explicitly using a Python function. A hybrid selector can be used for this purpose. If your selection model depends solely on genotype, you can define a function such as

```

def fitness_func(geno):
    # calculate fitness according to genotype at specified loci
    # genotypes are arrange locus by locus, namely A1,A2,B1,B2 for loci A and B
    return val
```

and use this function in an operator `PySelector(func=fitness_func, loci=loci)`. If your selection model depends on genotype as well as some information fields, you can define a function in the form of

```

def fitness_func(geno, field1, field2):
    # calculate fitness according to genotype at specified loci
    # and values at specified informaton fields.
    return val
```

where `field1`, `field2` are names of information fields. `simuPOP` will pass genotype and value of specified fields according to name of the passed function. Note that genotypes are arrange locus by locus, namely in the order of A1, 'A2', 'B1', 'B2' for loci A and B. Other parameters such as `gen`, `ind`, and `pop` are also allowed. Please check the reference manual for details.

When a `PySelector` is used to calculate fitness for an individual (parents if applied pre-mating, offspring if applied during-mating), it will collect his or her genotype at specified loci, optional values at specified information fields, generation number, or individual to a user-specified Python function, and take its return value as fitness. As you can imagine, the incorporation of information fields and generation number allow the implementation of very complex selection scenarios such as gene environment interaction and varying selection pressures.

Example `PySelector` demonstrates how to use a `PySelector` to specify fitness values according to a fitness table and the smoking status of each individual.

Example: A hybrid selector

```

>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=2000, loci=[1]*2, infoFields=['fitness', 'smoking'])
>>> s1 = .02
>>> s2 = .03
>>> # the second parameter gen can be used for varying selection pressure
>>> def sel(geno, smoking):
...     #      BB   Bb   bb
...     # AA   1    1    1
...     # Aa   1    1-s1 1-s2
...     # aa   1    1    1-s2
...     #
...     # geno is (A1 A2 B1 B2)
...     if geno[0] + geno[1] == 1 and geno[2] + geno[3] == 1:
...         v = 1 - s1 # case of AaBb
...     elif geno[2] + geno[3] == 2:
...         v = 1 - s2 # case of ??bb
...     else:
...         v = 1 # other cases
...     if smoking:
...         return v * 0.9
...     else:
...         return v
...
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=sim.PySelector(loci=[0, 1], func=sel),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         # set smoking status randomly
...         sim.InitInfo(lambda : random.randint(0,1), infoFields='smoking'),
...         sim.Stat(alleleFreq=[0, 1], step=20),
...         sim.PyEval(r"%0.4f\t%0.4f\n" % (alleleFreq[0][1], alleleFreq[1][1])),
...     ],
...     step=20)
0.4943      0.4890
0.4880      0.4285
0.4898      0.4073
50
now exiting runScriptInteractively...

```

[Download PySelector.py](#)

5.9.8 Multi-locus random fitness effects (operator PyMlSelector)

If the fitness of individuals is determined by fitness effects over a large number of loci, both *MlSelector* and *PySelector* are difficult to use because the former requires a large number of single-locus selectors, and the latter requires the processing long genome sequences. If the overall fitness can be determined by fitness effects of mutants, a *PyMlSelector* can be used. This operator

- Calls a user-provided call-back function for each locus with at least a mutant (non-zero allele). The function can

accept location and genotype so the fitness can be location and genotype dependent. The return value is cached so the function will be called only once for each locus-genotype pair.

- The fitness of each individual is determined by fitness values of loci with at least one mutant, using the same methods as operator *MlSelector*. This implicitly assumes that loci without any mutant have fitness value 1 and will not contribute to the final fitness value.

Example *PySelector* demonstrates how to use a *PyMlSelector* to implement a fitness model where each mutant has a random fitness drawn from a Gamma distribution. An additive model is used so a homozygote will have a fitness penalty that doubles that of a heterozygote. Because the fitness values of heterozygote and homozygote at each locus are requested separately, a class is used to store locus-specific s values.

The fitness value of each locus-genotype pair is outputted to a file, and it should be interesting to plot the distribution of allele frequency at each locus against the fitness values, because mutants that suffer from stronger negative natural selection are supposed to be rarer.

Example: *Random fitness effect*

```
>>> import simuOpt
>>> simuOpt.setOptions(quiet=True, alleleType='mutant')
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population(size=2000, loci=[10000], infoFields=['fitness'])
>>>
>>> class GammaDistributedFitness:
...     def __init__(self, alpha, beta):
...         self.coefMap = {}
...         self.alpha = alpha
...         self.beta = beta
...
...     def __call__(self, loc, alleles):
...         # because s is assigned for each locus, we need to make sure the
...         # same s is used for fitness of genotypes 01 (1-s) and 11 (1-2s)
...         # at each locus
...         if loc in self.coefMap:
...             s = self.coefMap[loc]
...         else:
...             s = random.gammavariate(self.alpha, self.beta)
...             self.coefMap[loc] = s
...         #
...         if 0 in alleles:
...             return 1. - s
...         else:
...             return 1. - 2.*s
...
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.AcgtMutator(rate=[0.00001], model='JC69'),
...         sim.PyMlSelector(GammaDistributedFitness(0.23, 0.185),
...             output='>>sel.txt'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(numOfSegSites=sim.ALL_AVAIL, step=50),
...         sim.PyEval(r"Gen: %2d #seg sites: %d\n" % (gen, numOfSegSites)",
...             step=50)
...     ],
...     gen = 201
```

(continues on next page)

(continued from previous page)

```

... )
Gen: 0 #seg sites: 180
Gen: 50 #seg sites: 1310
Gen: 100 #seg sites: 1479
Gen: 150 #seg sites: 1511
Gen: 200 #seg sites: 1579
201
>>> print(''.join(open('sel.txt').readlines()[:5]))
5855 1      0      0.978125
1085 2      0      0.340724
2907 0      1      0.998146
7773 0      1      0.927273
1835 0      2      0.999976

now exiting runScriptInteractively...

```

[Download PyMSelector.py](#)

5.9.9 Alternative implementations of natural selection

If you know how natural selection works in simuPOP, you do not have to use a selector to perform natural selection. For example,

- If you choose to use fitness values of parents to perform probabilistic natural selection during mating, you just need to set individual fitness in some way before mating. (You do not even have to use information field `fitness` because you can specify which information field to use in a mating scheme using parameter `selectionField`). This can be done through a penetrance model (as shown in the following example) where affected individuals are selected against during mating, a quantitative trait model (where a trait is defined to control individual fitness), or by setting information field fitness manually through a Python operator.
- If you would like to perform deterministic selection on certain phenotype, you can explicitly remove individuals before or during mating. More explicitly, you can use an operator *DiscardIf* to remove parents before mating or remove offspring during mating according to certain status (disease status or quantitative trait), provided that the trait status is defined before this operator is applied.

Example *penSelector* demonstrates a commonly used case where parents who are affected with certain disease are excluded from producing offspring. In this example, a penetrance model (operator *MaPenetrance*) is applied to the parental generation to determine who will be affected. An *InfoExec* operator is used to set individual fitness to 1 if he or she is unaffected, and 0 if he or she is affected. Due to the way parents are selected, affected parents will not be able to produce offspring as long as there is any unaffected individual. Because individual affection status is determined by his or her genotype, this genotype - affection status - fitness relationship could be implemented using an equivalent *MaSelector*. This method could be extended to *InfoExec*('fitness = 1 - 0.01*ind.affected()', exposeInd='ind') to select against, but not remove, affected parents, and similarly *InfoExec*('fitness = 1 - 0.01*(LDL > 250)') to select against individuals according to a quantitative trait. For this particular example, a *DiscardIf* operator could be used, although it can be slower because of the explicit removal of parents.

Example: *Natural selection according to individual affection status*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=1, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])

```

(continues on next page)

(continued from previous page)

```

...     ],
...     preOps=[
...         sim.MaPenetrance(loci=0, penetrance=[0.01, 0.1, 0.2]),
...         sim.Stat(numOfAffected=True, step=25, vars='propOfAffected'),
...         sim.PyEval(r'"Percent of affected: %.3f\t' % propOfAffected", step=50),
...         sim.InfoExec('fitness = not ind.affected()', exposeInd='ind')
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.PyEval(r'"%.4f\n' % alleleFreq[0][1]", step=50)
...     ],
...     gen=151
... )
Percent of affected: 0.110    0.4713
Percent of affected: 0.009    0.0095
Percent of affected: 0.013    0.0000
Percent of affected: 0.008    0.0000
151

now exiting runScriptInteractively...

```

[Download peneSelector.py](#)

5.9.10 Frequency dependent or dynamic selection pressure *

If individual fitness depends on individual information fields and/or population variables, you will have to calculate individual fitness using expressions or functions. In order to access individual information fields and population variable and calculate individual fitness, you have the option to

- Use a *PySelector* and pass genotype, values of information fields, references to individual and population to a user-provided function, which returns fitness value for each individual.
- Use of *PyOperator* to obtain information of the population (e.g. variables) and all individuals. Determine individual fitness and set information field `fitness` of all individuals.
- Use an operator *InfoExec* to calculate individual fitness using expressions. This method can be more efficient than others because simuPOP does not have to call a user-provided function.

Example *freqDependentSelection* demonstrates an example where the fitness values of individuals are calculated from allele frequencies calculated using a *Stat* operator. Because the fitness values of individuals are 1, , for genotype 00, 01 and 11 where is the frequency of allele 1, this allele will be under purifying selection if its frequency is over 0.5, and positive selection if its frequency is less than 0.5. Consequently, the frequency of this allele will oscillate around 0.5 during evolution, as shown in the result of this example.

Example: *Frequency dependent selection*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=2000, loci=1, infoFields='fitness')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=[
...         sim.Stat(alleleFreq=0),

```

(continues on next page)

(continued from previous page)

```

...     sim.InfoExec(''fitness = {
...         0: 1,
...         1: 1 - (alleleFreq[0][1] - 0.5)*0.1,
...         2: 1 - (alleleFreq[0][1] - 0.5)*0.2}[ind.allele(0,0)+ind.allele(0,1)]'
...     ),
...     exposeInd='ind'),
...     sim.Stat(meanOfInfo='fitness'),
...     sim.PyEval(r"alleleFreq=%.3f, mean fitness=%.5f\n" % (alleleFreq[0][1],
...     ↪meanOfInfo['fitness'])),
...     step=25),
... ],
... matingScheme=sim.RandomMating(),
... gen=151
... )
alleleFreq=0.495, mean fitness=1.00045
alleleFreq=0.504, mean fitness=0.99955
alleleFreq=0.484, mean fitness=1.00150
alleleFreq=0.492, mean fitness=1.00076
alleleFreq=0.499, mean fitness=1.00005
alleleFreq=0.526, mean fitness=0.99726
alleleFreq=0.514, mean fitness=0.99856
151

now exiting runScriptInteractively...

```

[Download freqDependentSelector.py](#)

5.9.11 Support for virtual subpopulations *

Support for virtual subpopulations allows you to use different selectors for different (virtual) subpopulations. Because virtual subpopulations may overlap, and they do not have to cover all individuals in a subpopulation, it is important to remember that

- If virtual subpopulations overlap, the fitness value set by the last selector will be used.
- If an individual is not included in any of the virtual subpopulation, its fitness value will be zero which will prevent them from producing any offspring.

Example *vspSelector* demonstrates how to apply selectors to virtual subpopulations. This example has two subpopulations, each having two virtual subpopulations defined by sex. Natural selection is applied to male individuals in the first subpopulation, and female individuals in the second subpopulation. However, because the sex of offspring is randomly determined, the selection actually decreases the disease allele frequency for all individuals.

Example: *Selector in virtual subpopulations*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[5000, 5000], loci=1, infoFields='fitness')
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     preOps=[
...         sim.MaSelector(loci=0, fitness=[1, 1, 0.98], subPops=[(0,0), (1,1)]),
...         sim.MaSelector(loci=0, fitness=[1, 0.99, 0.98], subPops=[(0,1), (1,0)]),

```

(continues on next page)

(continued from previous page)

```

...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=[0], subPops=[(sim.ALL_AVAIL, sim.ALL_AVAIL)],
...             vars='alleleFreq_sp', step=50),
...         sim.PyEval(r"%.4f\t%.4f\t%.4f\t%.4f\n" % "
...             tuple([subPop[x]['alleleFreq'][0][1] for x in ((0,0),(0,1),(1,0),(1,
↪1))])"),
...         step=50)
...     ],
...     gen=151
... )
0.5022      0.5083  0.4970  0.5020
0.4086      0.4054  0.3849  0.3817
0.3275      0.3259  0.2435  0.2532
0.2715      0.2662  0.1305  0.1338
151

now exiting runScriptInteractively...

```

Download vspSelector.py

Selecting through offspring can also be applied to virtual subpopulations. For example, Example *vspDuringMatingSelector* moves the selectors to the ops parameter of *RandomMating*. In this way, male and female offspring will have different survival probabilities according to their genotype.

Example: Selection against offspring in virtual subpopulations

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[5000, 5000], loci=1, infoFields='fitness')
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.5, .5])
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.MaSelector(loci=0, fitness=[1, 1, 0.98], subPops=[(0,0), (1,1)]),
...         sim.MaSelector(loci=0, fitness=[1, 0.99, 0.98], subPops=[(0,1), (1,0)]),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=[0], subPops=[(sim.ALL_AVAIL, sim.ALL_AVAIL)],
...             vars='alleleFreq_sp', step=50),
...         sim.PyEval(r"%.4f\t%.4f\t%.4f\t%.4f\n" % "
...             tuple([subPop[x]['alleleFreq'][0][1] for x in ((0,0),(0,1),(1,0),(1,
↪1))])"),
...         step=50)
...     ],
...     gen=151
... )
0.5018      0.5034  0.4941  0.4853
0.3652      0.3728  0.3820  0.3766
0.2882      0.2920  0.2590  0.2667
0.2083      0.1994  0.2378  0.2356
151

```

(continues on next page)

(continued from previous page)

```
now exiting runScriptInteractively...
```

Download [vspDuringMatingSelector.py](#)

5.9.12 Natural selection in heterogeneous mating schemes **

Multiple mating schemes could be applied to the same subpopulation in a heterogeneous mating scheme (*HeteroMating*). These mating schemes may or may not support natural selection, may be applied to different virtual subpopulations of population, and they may see Individuals differently in terms of individual fitness. Parameter `fitnessField` of a mating scheme could be used to handle such cases. More specifically,

- You can turn off the natural selection support of a mating scheme by setting `fitnessField=''`.
- If a mating scheme uses a different set of fitness values, you can add an information field (e.g. `fitness1`), setting individual fitness to this information field using a selector (with parameter `infoFields='fitness1'`) and tells a mating scheme to look in this information field for fitness values (using parameter `fitnessField='fitness1'`).

5.10 Tagging operators

In simuPOP, tagging refers to the action of setting various information fields of offspring, usually using various parental information during the production of offspring. simuPOP provides a number of tagging operators (called taggers) for various purposes. Because tagging operators are during-mating operators, parameter `subPops` can be used to tag only offspring that belong to specified virtual subpopulation. (e.g. all male offspring)

5.10.1 Inheritance tagger (operator `InheritTagger`)

An inheritance tagger passes values of parental information field(s) to the corresponding offspring information field(s). Depending on the parameters, an `InheritTagger` can

- For asexual mating schemes, pass one or more information fields from parent to offspring.
- Pass one or more information fields from father to offspring (`mode=PATERNAL`).
- Pass one or more information fields from mother to offspring (`mode=MATERNAL`).
- Pass the maximal, minimal, sum, multiplication or average of values of one or more information fields of both parents (`mode=MAXIMUM, MINIMUM, ADDITION, MULTIPLICATION` or `MEAN`).

This can be used to track the spread of certain information during evolution. For example, Example *InheritTagger* tags the first individuals of ten subpopulations of size 1000. individuals in the offspring generation inherits the maximum value of field `x` from his/her parents so `x` is inherited regardless of the sex of parents. A Stat operator is used to calculate the number of offspring having this tag in each subpopulation. The results show that some tagged ancestors have many offspring, and some have none. If you run this simulation long enough, you can see that all ancestors become the ancestor of either none or all individuals in a population. Note that this simulation only considers genealogical inheritance and ancestors do not have to pass any genotype to the last generation.

Example: *Use an inherit tagger to track offspring of individuals*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000]*10, loci=1, infoFields='x')
>>> # tag the first individual of each subpopulation.
>>> for sp in range(pop.numSubPop()):
```

(continues on next page)

(continued from previous page)

```

...     pop.individual(0, sp).x = 1
...
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.InheritTagger(mode=sim.MAXIMUM, infoFields='x'),
...     ]),
...     postOps=[
...         sim.Stat(sumOfInfo='x', vars=['sumOfInfo_sp']),
...         sim.PyEval(r' ".join(["%3d" % subPop[i]["sumOfInfo"]["x"] for i in_
↳ range(10)])+"\n"',
...     ],
...     gen = 5
... )
2, 1, 0, 1, 1, 2, 3, 3, 1, 1
5, 1, 0, 1, 1, 3, 3, 5, 3, 0
9, 2, 0, 2, 2, 7, 9, 5, 13, 0
21, 4, 0, 2, 5, 18, 11, 9, 27, 0
39, 5, 0, 6, 8, 36, 23, 20, 67, 0
5
now exiting runScriptInteractively...
```

[Download InheritTagger.py](#)

5.10.2 Summarize parental informatin fields (operator SummaryTagger)

A *SummaryTagger* summarize values of one or more parental information fields and place the result in an offspring information field. If mating is sexual, two sets of values will be involved. Summarization methods include MEAN, MINIMUM, MAXIMUM, SUMMATION and MULTIPLICATION. The operator is usually used to summarize certain characteristic of parents of each offspring. For example, a *SummaryTagger* is used in Example *SummaryTagger* to calculate the mean fitness of parents during each mating event. The results are saved in the avgFitness field of offspring. Because allele 1 at locus 0 is under purifying selection, the allele frequency of this allele decreases. In the mean time, fitness of parents increases because less and less parents have this allele.

Example: *Using a summary tagger to calculate mean fitness of parents.*

```

>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1, infoFields=['fitness', 'avgFitness'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     preOps=sim.MaSelector(loci=0, wildtype=0, fitness=[1, 0.99, 0.95]),
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.SummaryTagger(mode=sim.MEAN, infoFields=['fitness', 'avgFitness']),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=0, meanOfInfo='avgFitness', step=10),
...         sim.PyEval(r'"gen %d: allele freq: %.3f, average fitness of parents: %.
↳ 3f\n' % "
...         "(gen, alleleFreq[0][1], meanOfInfo['avgFitness'])", step=10)
```

(continues on next page)

(continued from previous page)

```

...     ],
...     gen = 50,
... )
gen 0: allele freq: 0.473, average fitness of parents: 0.984
gen 10: allele freq: 0.421, average fitness of parents: 0.986
gen 20: allele freq: 0.388, average fitness of parents: 0.988
gen 30: allele freq: 0.288, average fitness of parents: 0.991
gen 40: allele freq: 0.256, average fitness of parents: 0.993
50

now exiting runScriptInteractively...
```

[Download SummaryTagger.py](#)

5.10.3 Tracking parents (operator `ParentsTagger`)

A parents tagger is used to record the indexes of parents (in the parental population) in the information fields (default to `father_idx`, `mother_idx`) of their offspring. These indexes provide a way to track down an individual's parents, offspring and consequently all relatives in a multi-generation population. Because this operator has been extensively used in this guide, please refer to other sections for an Example (e.g. Example *basicInfoFields*).

As long as parental generations do not change after the offspring generation is created, recorded parental indexes can be used to locate parents of an individual. However, in certain applications when parental generations change (e.g. to draw a pedigree from a large population), or when individuals can not be looked up easily using indexes (e.g. after individuals are saved to a file), giving every Individual an unique ID and refer to them using ID will be a better choice.

5.10.4 Tracking index of offspring within families (operator `OffspringTagger`)

An offspring tagger is used to record the index of offspring within each family in an information field (default to `offspring_idx`) of offspring. Because the index is reset for each mating event, the index will be reset even if two adjacent families share the same parents. In addition, this operator records the relative index of an offspring so the index will not change if an offspring is re-generated when the previous offspring is discarded for any reason.

Because during-mating selection operator discards offspring according to their genotypes, a mating scheme can produce families with varying sizes even if `numOffspring` is set to a constant number. On the other hand, if we would like to ensure equal family size N in the presence of natural selection, we will have to produce more offspring so that there can be at least N offspring in each family after selection. Once N offspring have been generated, excessive offspring can be discarded according to `offspring_idx`. The following example demonstrates such a simulation scenario:

Example: *Keeping constant family size in the presence of natural selection against offspring*

```

>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=1, infoFields='offspring_idx')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         # lethal recessive alleles
...         sim.MaSelector(loci=0, wildtype=0, fitness=[1, 0.90, 0.5]),
...         sim.OffspringTagger(),
...         sim.DiscardIf('offspring_idx > 4'),
...     ])
```

(continues on next page)

(continued from previous page)

```

...     ], numOffspring=10),
...     postOps=[
...         sim.Stat(alleleFreq=0, step=10),
...         sim.PyEval(r'"gen %d: allele freq: %.3f\n' % "
...             "(gen, alleleFreq[0][1])", step=10)
...     ],
...     gen = 50,
... )
gen 0: allele freq: 0.445
gen 10: allele freq: 0.187
gen 20: allele freq: 0.089
gen 30: allele freq: 0.087
gen 40: allele freq: 0.059
50
now exiting runScriptInteractively...
```

Download OffspringTagger.py

Because families with lethal alleles produce the same number of offspring as families without such alleles, natural selection happens within each families and is weaker than the case when natural selection is used to all offspring. This phenomena is generally referred to as reproductive compensation.

5.10.5 Assign unique IDs to individuals (operator *IdTagger*)

Although it is possible to use generation number and individual indexes to locate individuals in an evolving population, an unique ID makes it much easier to identify individuals when migration is involved, and to analyze an evolutionary process outside of simuPOP. An operator *IdTagger* (and its function form *tagID*) is provided by simuPOP to assign an unique ID to all individuals during evolution.

The IDs of individuals are usually stored in an information field named *ind_id*. To ensure uniqueness across populations, a single source of ID is used for this operator. individual IDs are assigned consecutively starting from 0. If you would like to reset the sequence or start from a different number, you can call the *reset(startID)* function of any *IdTagger*.

An *IdTagger* is usually used during-mating to assign ID to each offspring. However, if it is applied directly to a population, it will assign unique IDs to all individuals in this population. This property is usually used in the *preOps* parameter of function *Simulator.evolve* to assign initial ID to a population. For example, two *IdTagger* operators are used in Example *IdTagger* to assign IDs to all individuals. Although different operators are used, different IDs are assigned to individuals.

Example: Assign unique IDs to individuals

```

>>> import simuPOP as sim
>>> pop = sim.Population(10, infoFields='ind_id', ancGen=1)
>>> pop.evolve(
...     initOps=sim.IdTagger(),
...     matingScheme=sim.RandomSelection(ops=[
...         sim.CloneGenoTransmitter(),
...         sim.IdTagger(),
...     ]),
...     gen = 1
... )
1
>>> print([int(ind.ind_id) for ind in pop.individuals()])
[11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

(continues on next page)

(continued from previous page)

```

>>> pop.useAncestralGen(1)
>>> print([int(ind.ind_id) for ind in pop.individuals()])
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> sim.tagID(pop) # re-assign ID
>>> print([int(ind.ind_id) for ind in pop.individuals()])
[21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

now exiting runScriptInteractively...

```

[Download IdTagger.py](#)

5.10.6 Tracking Pedigrees (operator PedigreeTagger)

A *PedigreeTagger* is similar to a *ParentsTagger* in that it records parental information in offspring's information fields. However, instead of indexes of parents, this operator records a unique ID of each parent to make it easier to study and reconstruct a complete pedigree of a whole evolutionary process. The default information fields are `father_id` and `mother_id`.

By default, the *PedigreeTagger* does not produce any output. However, if a valid output string (or function) is specified, it will output the ID of offspring and their parents, sex and affection status of offspring, and optionally values at specified information fields (parameter `outputFields`) and genotype at specified loci (parameter `outputLoci`). Because this operator only outputs offspring, the saved file does not have detailed information of individuals in the top-most ancestral generation. If you would like to record complete pedigree information, you can apply *PedigreeTagger* in the `initOps` operator of function *Simulator.evolve* or *Population.evolve* to output information of the initial population. Although this operator is primarily used to output pedigree information, values at specified information fields and genotypes at specified loci could also be outputted.

Example *PedigreeTagger* demonstrates how to output the complete pedigree of an evolutionary process. Note that *IdTagger* has to be applied before *PedigreeTagger* so that IDs of offspring could be assigned before they are outputted.

Example: *Output a complete pedigree of an evolutionary process*

```

>>> import simuPOP as sim
>>> pop = sim.Population(100, infoFields=['ind_id', 'father_id', 'mother_id'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...         sim.PedigreeTagger(output='>>pedigree.txt'),
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.IdTagger(),
...         sim.PedigreeTagger(output='>>pedigree.txt'),
...         sim.MendelianGenoTransmitter())
...     ),
...     gen = 100
... )
100
>>> ped = open('pedigree.txt')
>>> lines = ped.readlines()
>>> ped.close()
>>> # first few lines, saved by the first PedigreeTagger
>>> print(''.join(lines[:3]))
1 0 0 F U

```

(continues on next page)

(continued from previous page)

```

2 0 0 F U
3 0 0 M U

>>> # last several lines, saved by the second PedigreeTagger
>>> print(''.join(lines[-3:]))
10098 9974 9915 F U
10099 9967 9997 M U
10100 9945 9936 M U

>>> # load this file
>>> ped = sim.loadPedigree('pedigree.txt')
>>> # should have 100 ancestral generations (plus one present generation)
>>> ped.ancestralGens()
100

now exiting runScriptInteractively...
```

[Download PedigreeTagger.py](#)

5.10.7 A hybrid tagger (operator PyTagger)

A *PyTagger* uses a user-defined function to pass parental information fields to offspring. When a mating event happens, this operator collect values of specified information fields of parents, pass them to a user-provided function, and use the return values to set corresponding offspring information fields. A typical usage of this operator is to set random environmental factors that are affected by parental values. Example *PyTagger* demonstrates such an example where the location of each offspring (x, y) is randomly assigned around the middle position of his or her parents.

Example: Use of a hybrid tagger to pass parental information to offspring

```

>>> import simuPOP as sim
>>> import random
>>> def randomMove(x, y):
...     '''Pass parental information fields to offspring'''
...     # shift right with high concentration of alleles...
...     off_x = random.normalvariate((x[0]+x[1])/2., 0.1)
...     off_y = random.normalvariate((y[0]+y[1])/2., 0.1)
...     return off_x, off_y
...
>>> pop = sim.Population(1000, loci=[1], infoFields=['x', 'y'])
>>> pop.setVirtualSplitter(sim.GenotypeSplitter(loci=0, alleles=[[0, 0], [0,1], [1,
↪1]]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.InitInfo(random.random, infoFields=['x', 'y'])
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.PyTagger(func=randomMove),
...     ]),
...     postOps=[
...         sim.Stat(minOfInfo='x', maxOfInfo='x'),
...         sim.PyEval(r'''Range of x: %.2f, %.2f\n' % (minOfInfo['x'], maxOfInfo['x'])
↪")
...     ])

```

(continues on next page)

(continued from previous page)

```

...     ],
...     gen = 5
... )
Range of x: -0.17, 1.12
Range of x: -0.05, 1.14
Range of x: 0.01, 1.01
Range of x: 0.01, 1.04
Range of x: 0.06, 0.95
5
>>>

now exiting runScriptInteractively...

```

[Download PyTagger.py](#)

5.10.8 Tagging that involves other parental information

If the way how parental information fields pass to their offspring is affected by parental genotype, sex, or affection status, you could use a Python operator (*PyOperator*) during mating to explicitly obtain parental information and set offspring information fields.

Alternatively, you can add another information field, translate needed information to this field and pass the genotype information in the form of information field. Operator *InfoExec* could be helpful in this case. Example *otherTagging* demonstrates such an example where the number of affected parents are recorded in an information field. Before mating happens, a penetrance operator is used to assign affection status to parents. The affection status is then copied to an information field affected so that operator *SummaryTagger* could be used to count the number of affected parents. Two *MaPenetrance* operators are used both before and after mating to assign affection status to both parental and offspring generations. This helps dividing the offspring generation into affected and unaffected virtual subpopulations. Not surprisingly, the average number of affected parents is larger for affected individuals than unaffected individuals.

Example: *Tagging that involves other parental information*

```

>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[1], infoFields=['aff', 'numOfAff'])
>>> # define virtual subpopulations by affection sim.status
>>> pop.setVirtualSplitter(sim.AffectionSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     preOps=[
...         # get affection sim.status for parents
...         sim.MaPenetrance(loci=0, wildtype=0, penetrance=[0.1, 0.2, 0.4]),
...         # set 'aff' of parents
...         sim.InfoExec('aff = ind.affected()', exposeInd='ind'),
...     ],
...     # get number of affected parents for each offspring and store in numOfAff
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.SummaryTagger(mode=sim.SUMMATION, infoFields=['aff', 'numOfAff'])]),
...     postOps=[
...         # get affection sim.status for offspring
...         sim.MaPenetrance(loci=0, wildtype=0, penetrance=[0.1, 0.2, 0.4]),
...         # calculate mean 'numOfAff' of offspring, for unaffected and affected_
...         # subpopulations.

```

(continues on next page)

(continued from previous page)

```

...     sim.Stat(meanOfInfo='numOfAff', subPops=[(0,0), (0,1)], vars=['meanOfInfo_
↳sp']),
...     # print mean number of affected parents for unaffected and affected_
↳offspring.
...     sim.PyEval(r"Mean number of affected parents: %.2f (unaff), %.2f (aff)\n
↳' % "
...         "(subPop[(0,0)]['meanOfInfo']['numOfAff'], subPop[(0,1)]['meanOfInfo
↳']['numOfAff'])")
...     ],
...     gen = 5
... )
Mean number of affected parents: 0.41 (unaff), 0.44 (aff)
Mean number of affected parents: 0.41 (unaff), 0.54 (aff)
Mean number of affected parents: 0.47 (unaff), 0.55 (aff)
Mean number of affected parents: 0.47 (unaff), 0.55 (aff)
Mean number of affected parents: 0.42 (unaff), 0.45 (aff)
5
>>>

now exiting runScriptInteractively...
```

[Download otherTagging.py](#)

5.11 Statistics calculation (operator Stat)

5.11.1 How statistics calculation works

A *Stat* operator calculates specified statistics of a population when it is applied to this population. This operator can be applied to specified replicates (parameter *rep*) at specified generations (parameter *begin*, *end*, *step*, and *at*). This operator does not produce any output (ignore parameter *output*) after statistics are calculated. Instead, it stores results in the local namespace of the population being applied. Other operators can retrieve these variables or evaluate expression directly in this local namespace.

The *Stat* operator is usually used in conjunction with a *PyEval* or *PyExec* operator which execute Python statements and/or expressions in a population's local namespace. For example, operators

```
ops = [
    Stat(alleleFreq=[0]),
    PyEval("'%.2f' % alleleFreq[0][0]")
]
```

in the *ops* parameter of the *Simulator.evolve* function will be applied to populations during evolution. The first operator calculates allele frequency at the first locus and store the results in each population's local namespace. The second operator formats and outputs one of the variables. Because of the flexibility of the *PyEval* operator, you can output statistics, even simple derived statistics, in any format. For example, you can output expected heterozygosity (H_{exp}) using calculated allele frequencies as follows:

```
PyEval("'H_exp=%.2f' % (1-sum([x*x for x in alleleFreq[0].values()])))")
```

Note that `alleleFreq[0]` is a dictionary.

You can also retrieve variables in a population directly using functions *Population.vars()* or *Population.dvars()*. The only difference between these functions is that *vars* returns a dictionary and *dvars* returns a Python

object that uses variable names as attributes (`vars()['alleleFreq']` is equivalent to `dvars.alleleFreq`). This method is usually used when the function form of the *Stat* operator is used. For example,

```
stat(pop, alleleFreq=[0])
H_exp = 1 - sum([x*x for x in pop.dvars().alleleFreq[0].values()])
```

uses the `stat` function (note the capital S) to count frequencies of alleles for a given population and calculates expected heterozygosity using these variables.

5.11.2 defdict datatype

simuPOP uses dictionaries to save statistics such as allele frequencies. For example, `alleleFreq[5]` can be `{0:0.2, 3:0.8}` meaning there are 20% allele 0 and 80% allele 3 at locus 5 in a population. However, because it is sometimes unclear whether or not a particular allele exists in a population, `alleleFreq[5][allele]` can fail with a `KeyError` exception if `alleleFreq[5]` does not have key `allele`.

To address this problem, a special default dictionary type `defdict` is used for dictionaries with keys determined from a population. This derived dictionary type works just like a regular dictionary, but it returns 0, instead of raising a `KeyError` exception, when an invalid key is used. For example, subpopulations in Example *defdictType* have different alleles. Although `pop.dvars(sp).alleleFreq[0]` have only two keys for `sp=0` or `1`, `pop.dvars(sp).alleleFreq[0][x]` are used to print frequencies of alleles 0, 1 and 2.

Example: *The defdict datatype*

```
>>> import simuPOP as sim
>>> pop = sim.Population([100]*2, loci=1)
>>> sim.initGenotype(pop, freq=[0, 0.2, 0.8], subPops=0)
>>> sim.initGenotype(pop, freq=[0.2, 0.8], subPops=1)
>>> sim.stat(pop, alleleFreq=0, vars=['alleleFreq_sp'])
>>> for sp in range(2):
...     print('Subpop %d (with %d alleles): ' % (sp, len(pop.dvars(sp).
↪alleleFreq[0])))
...     for a in range(3):
...         print('%2f ' % pop.dvars(sp).alleleFreq[0][a])
...
Subpop 0 (with 2 alleles):
0.00
0.21
0.79
Subpop 1 (with 2 alleles):
0.21
0.79
0.00

now exiting runScriptInteractively...
```

[Download defdict.py](#)

Note: The standard collections module of Python has a `defaultdict` type that accepts a default factory function that will be used when an invalid key is encountered. The `defdict` type is similar to `defaultdict(int)` but with an important difference: when an invalid key is encountered, `d[key]` with a default value will be inserted to a `defaultdict(int)`, but will not be inserted to a `defdict`. That is to say, it is safe to use `alleleFreq[loc].keys()` to get available alleles after non-assignment `alleleFreq[loc][allele]` operations.

5.11.3 Support for virtual subpopulations

The *Stat* operator supports parameter *subPops* and can calculate statistics in specified subpopulations. For example

```
Stat(alleleFreq=[0], subPops=[(0, 0), (1, 0)])
```

will calculate the frequencies of alleles at locus 0, among Individuals in two virtual subpopulations. If the virtual subpopulation is defined by sex (using a *SexSplitter*), the above operator will calculate allele frequency among all males in the first and second subpopulations (not separately!). If *subPops* is not specified, allele frequency of the whole population (all subpopulations) will be calculated.

Although many statistics could be calculated and outputted, the *Stat* operator by default outputs a selected number of variables for each statistic calculated. Other statistics could be calculated and outputted if their names are specified in parameter *vars*. Variable names ending with *_sp* is interpreted as variables that will be calculated and outputted in all or specified (virtual) subpopulations. For example, parameter *vars* in

```
Stat(alleleFreq=[0], subPops=[0, (1, 0)], vars=['alleleFreq_sp', 'alleleNum_sp'])
```

tells this operator to output numbers and frequencies of alleles at locus 0 in subpopulation 0 and virtual subpopulation (1, 0). These variables will be saved in dictionaries *subPop[sp]* of the local namespace. For example, the above operator will write variables such as *subPop[0]['alleleFreq']*, *subPop[(1,0)]['alleleFreq']* and *subPop[(1,0)]['alleleNum']*. Functions *Population.vars(sp)* and *Population.dvars(sp)* are provided as shortcuts to access these variables but the full variable names have to be specified if these variables are used in expressions.

By default, the same variables will be set for a statistic, regardless of the values of the *loci* and *subPops* parameter. This can be a problem if multiple *Stat* operators are used to calculate the same statistics for different sets of loci (e.g. for each chromosome) or subpopulations. To avoid name conflict, you can use parameter *suffix* to add a suffix to all variables outputted by a *Stat* operator. For example, Example *statSuffix* uses 4 *Stat* operators to calculate overall and pairwise values for three subpopulations. Different suffixes are used for pairwise estimators so that variables set by these operators will not override each other.

Example: Add suffixes to variables set by multiple *Stat* operators

```
>>> import simuPOP as sim
>>> pop = sim.Population([5000]*3, loci=5)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(structure=range(5), subPops=(0, 1), suffix='_01', step=40),
...         sim.Stat(structure=range(5), subPops=(1, 2), suffix='_12', step=40),
...         sim.Stat(structure=range(5), subPops=(0, 2), suffix='_02', step=40),
...         sim.Stat(structure=range(5), step=40),
...         sim.PyEval(r"'Fst=%.3f (pairwise: %.3f %.3f %.3f)\n' % (F_st, F_st_01, F_
↪st_12, F_st_02)",
...             step=40),
...     ],
...     gen = 200
... )
Fst=0.000 (pairwise: 0.000 0.000 0.000)
Fst=0.004 (pairwise: 0.006 0.003 0.004)
Fst=0.012 (pairwise: 0.017 0.015 0.004)
Fst=0.008 (pairwise: 0.012 0.010 0.001)
```

(continues on next page)

(continued from previous page)

```
Fst=0.008 (pairwise: 0.007 0.009 0.007)
200

now exiting runScriptInteractively...
```

[Download statSuffix.py](#)

Note: The `Stat` operator accepts overlapping or even duplicate virtual subpopulations. During the calculation of summary statistics, these subpopulations are treated as separate subpopulations so some individuals can be counted more than once. For example, individuals in virtual subpopulation (0, 1) will be counted twice during the calculation of allele frequency and population size in operator

```
Stat(alleleFreq=[0], popSize=True, subPops=[0, (0, 1)])
```

5.11.4 Counting individuals by sex and affection status

Parameters `popSize`, `numOfMales` and `numOfAffected` provide basic Individual counting statistics. They count the number of all, male/female, affected/unaffected individuals in all or specified (virtual) subpopulations, and set variables such as `popSize`, `numOfMales`, `numOfFemales`, `numOfAffected`, `numOfUnaffected`. Proportions and statistics for subpopulations are available if variables such as `propOfMales`, `numOfAffected_sp` are specified in parameter vars. Another variable `subPopSize` is defined for parameter `popSize=True`. It is a list of sizes of all or specified subpopulations and is easier to use than referring to variable `popSize` from individual subpopulations.

Example `statCount` demonstrates how to use these parameters in operator `Stat`. It defines four VSPs by sex and affection status (using a `stackedSplitter`) and count individuals by sex and affection status. It is worth noting that `pop.dvars().popSize` in the first example is the total number of individuals in two virtual subpopulations (0, 0) and (0, 2), which are all male individuals, and all unaffected individuals. Because these two VSPs overlap, this variable can be larger than actual population size.

Example: *Count individuals by sex and/or affection status*

```
>>> import simuPOP as sim
>>> pop = sim.Population(10000, loci=1)
>>> pop.setVirtualSplitter(sim.CombinedSplitter(
...     [sim.SexSplitter(), sim.AffectionSplitter()]))
>>> sim.initSex(pop)
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> sim.maPenetrance(pop, loci=0, penetrance=[0.1, 0.2, 0.5])
>>> # Count sim.population size
>>> sim.stat(pop, popSize=True, subPops=[(0, 0), (0, 2)])
>>> # popSize is the size of two VSPs, does not equal to total sim.population size.
>>> # Because two VSPs overlap (all males and all unaffected), popSize can be
>>> # greater than real sim.population size.
>>> print(pop.dvars().subPopSize, pop.dvars().popSize)
[5052, 6080] 11132
>>> # print popSize of each virtual subpopulation.
>>> sim.stat(pop, popSize=True, subPops=[(0, 0), (0, 2)], vars='popSize_sp')
>>> # Note the two ways to access variable in (virtual) subpopulations.
>>> print(pop.dvars((0,0)).popSize, pop.dvars().subPop[(0,2)]['popSize'])
5052 6080
>>> # Count number of male (should be the same as the size of VSP (0,0)).
>>> sim.stat(pop, numOfMales=True)
>>> print(pop.dvars().numOfMales)
```

(continues on next page)

(continued from previous page)

```

5052
>>> # Count the number of affected and unaffected male individual
>>> sim.stat(pop, numOfMales=True, subPops=[(0, 2), (0, 3)], vars='numOfMales_sp')
>>> print(pop.dvars((0,2)).numOfMales, pop.dvars((0,3)).numOfMales)
3056 1996
>>> # or number of affected male and females
>>> sim.stat(pop, numOfAffected=True, subPops=[(0, 0), (0, 1)], vars='numOfAffected_sp
↳')
>>> print(pop.dvars((0,0)).numOfAffected, pop.dvars((0,1)).numOfAffected)
1996 1924
>>> # These can also be done using a sim.ProductSplitter...
>>> pop.setVirtualSplitter(sim.ProductSplitter(
...     [sim.SexSplitter(), sim.AffectionSplitter()])))
>>> sim.stat(pop, popSize=True, subPops=[(0, x) for x in range(4)])
>>> # counts for male unaffected, male affected, female unaffected and female affected
>>> print(pop.dvars().subPopSize)
[3056, 1996, 3024, 1924]

now exiting runScriptInteractively...

```

[Download statCount.py](#)

5.11.5 Number of segregating and fixed sites

Parameter *numOfSegSites* counts the number of segregating sites for specified or all loci, for all individuals or individuals in specified (virtual) subpopulations. It can also be used to count the number of fixed sites. This parameter sets variables *numOfSegSites* and *numOfFixedSites*. Here we defined fixed sites as loci with only one non-zero allele (e.g. fixed to a non-zero allele). Other numbers, such as all loci with only one allele (including zero), or loci with all wildtype alleles (only zero), can be derived from these two counts. Starting from version 1.1.3, variables *segSites* and *fixedSites* can be used to return a list of segregating and fixed sites.

For example, Example *numSegSites* demonstrates how to use this operator to calculate the number of segregating sites (sites with alleles 0 and 1), number of fixed sites (sites with only allele 1), and number of loci with only wildtype alleles (loci with only allele 0). As you can see, the population starts with 100 segregating sites. During evolution, alleles at some loci get lost and some get fixed, and there should be no segregating site if we evolve the population for long enough.

Example: *Count number of segregating and fixed sites*

```

>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=[1]*100)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...         sim.PyOutput('#all 0\t#seg sites\t#all 1\n'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(numOfSegSites=sim.ALL_AVAIL,
...                 vars=['numOfSegSites', 'numOfFixedSites']),
...         sim.PyEval(r'"%d\t%d\t%d\n" % (100-numOfSegSites-numOfFixedSites, '
...                 'numOfSegSites, numOfFixedSites)',
...                 step=50)
...     ],

```

(continues on next page)

(continued from previous page)

```

...     gen=500
... )
#all 0      #seg sites      #all 1
0    100     0
0    93      7
3    76     21
7    55     38
12   40     48
17   31     52
19   23     58
22   19     59
26   14     60
28   10     62
500
>>> # output a list of segregating sites
>>> sim.stat(pop, numOfSegSites=sim.ALL_AVAIL, vars='segSites')
>>> print(pop.dvars().segSites)
[11, 15, 20, 32, 39, 43, 44, 51, 86, 95]

now exiting runScriptInteractively...

```

[Download statNumOfSegSites.py](#)

5.11.6 Allele count and frequency

Parameter *alleleFreq* accepts a list of markers at which allele frequencies in all or specified (virtual) subpopulations will be calculated. This statistic sets variables `alleleFreq[loc][allele]` and `alleleNum[loc][allele]` which are frequencies and numbers of allele *allele* at locus *loc*, respectively. If variables `alleleFreq_sp` and `alleleNum_sp` are specified in parameter *vars*, these variables will be set for all or specified (virtual) subpopulations. **At the Python level, these variables are dictionaries of default dictionaries.** That is to say, `alleleFreq[loc]` at an unspecified locus will raise a `KeyError` exception, and `alleleFreq[loc][allele]` of an invalid allele will return 0.

Example *statAlleleFreq* demonstrates an advanced usage of allele counting statistic. In this example, two virtual subpopulations are defined by individual affection status. During evolution, a multi-allele penetrance operator is used to determine individual affection status and a *Stat* operator is used to calculate allele frequencies in these two virtual subpopulations, and in the whole population. Because the simulated disease is largely caused by the existence of allele 1 at the first locus, it is expected that the frequency of allele 1 is higher in the case group than in the control group. It is worth noting that `alleleFreq[0][1]` in this example is the frequency of allele 1 in the whole population because these two virtual subpopulations add up to the whole population.

Example: Calculate allele frequency in affected and unaffected individuals

```

>>> import simuPOP as sim
>>> pop = sim.Population(10000, loci=1)
>>> pop.setVirtualSplitter(sim.AffectionSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(loci=0, freq=[0.8, 0.2])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.MaPenetrance(penetrance=[0.1, 0.4, 0.6], loci=0),
...         sim.Stat(alleleFreq=0, subPops=[(0, 0), (0, 1)]),
...     ]
... )

```

(continues on next page)

(continued from previous page)

```

...         vars=['alleleFreq', 'alleleFreq_sp']),
...         sim.PyEval(r"Gen: %d, freq: %.2f, freq (aff): %.2f, freq (unaff): %.2f\n
→ ' % " + \
...             "(gen, alleleFreq[0][1], subPop[(0,1)]['alleleFreq'][0][1], " + \
...             "subPop[(0,0)]['alleleFreq'][0][1])"),
...     ],
...     gen = 5
... )
Gen: 0, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 1, freq: 0.20, freq (aff): 0.40, freq (unaff): 0.14
Gen: 2, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 3, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 4, freq: 0.19, freq (aff): 0.41, freq (unaff): 0.14
5
now exiting runScriptInteractively...

```

[Download statAlleleFreq.py](#)

5.11.7 Genotype count and frequency

Parameter *genoFreq* accepts a list of loci at which genotype counts and frequencies are calculated and outputted. A genotype is represented as a tuple of alleles at a locus. The length of the tuples** is determined by the number of homologous copy of chromosomes in a population. For example, genotypes in a diploid population are ordered pairs such as (1, 2) where 1 and 2 are alleles at a locus on, respectively, the first and second homologous copies of chromosomes. (1, 2) and (2, 1) are different genotypes. This statistic sets dictionaries (with locus indexes as keys) of default dictionaries (with genotypes as keys) *genoFreq* and *genoNum*.

Example *statGenoFreq* creates a small population and initializes a locus with rare alleles 0, 1 and a common allele 2. A function *stat* (the function form of operator *Stat*) is used to count the available genotypes. Note that `pop.dvars().genoFreq[0][(i, j)]` can be used to print frequencies of all genotypes even when not all genotypes are available in the population.

Example: *Counting genotypes in a population*

```

>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=[1, 1, 1], lociNames=['A', 'X', 'Y'],
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y])
>>> sim.initGenotype(pop, freq=[0.01, 0.05, 0.94])
>>> sim.stat(pop, genoFreq=['A', 'X']) # both loci indexes and names can be used.
>>> print('Available genotypes on autosome:', list(pop.dvars().genoFreq[0].keys()))
Available genotypes on autosome: [(0, 2), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> for i in range(3):
...     for j in range(3):
...         print('%d-%d: %.3f' % (i, j, pop.dvars().genoFreq[0][(i, j)]))
...
0-0: 0.000
0-1: 0.000
0-2: 0.020
1-0: 0.000
1-1: 0.030
1-2: 0.070
2-0: 0.010
2-1: 0.040
2-2: 0.830

```

(continues on next page)

(continued from previous page)

```
>>> print('Genotype frequency on chromosome X:\n', \
...       '\n'.join(['%s: %.3f' % (x,y) for x,y in pop.dvars().genoFreq[1].items()]))
Genotype frequency on chromosome X:
(0,): 0.020
(1,): 0.030
(2,): 0.950

now exiting runScriptInteractively...
```

[Download statGenoFreq.py](#)

5.11.8 Homozygote and heterozygote count and frequency

In a diploid population, a heterozygote is a genotype with two different alleles and a homozygote is a genotype with two identical alleles. Parameter `heteroFreq` accepts a list of loci and outputs variables `heteroFreq` which is a dictionary of heterozygote frequencies at specified loci. Optional variables `heteroNum`, `homoFreq` and `homoNum` can be outputted for all and each (virtual) subpopulations. Example *statHeteroFreq* demonstrates the decay of heterozygosity of a locus due to genetic drift.

Example: *Counting homozygotes and heterozygotes in a population*

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(heteroFreq=0, step=10),
...         sim.PyEval(r"Gen: %d, HeteroFreq: %.2f\n" % (gen, heteroFreq[0])),
...     ],
...     step=20)
Gen: 0, HeteroFreq: 0.45
Gen: 20, HeteroFreq: 0.44
Gen: 40, HeteroFreq: 0.55
Gen: 60, HeteroFreq: 0.46
Gen: 80, HeteroFreq: 0.40
100

now exiting runScriptInteractively...
```

[Download statHeteroFreq.py](#)

5.11.9 Haplotype count and frequency

Haplotypes refer to alleles on the same homologous copy of a chromosome at specified loci. For example, an diploid individual can have haplotypes (0, 2, 1) and (0, 1, 1) at loci (2, 3, 5) if he or she has genotype (0, 0), (2, 1) and (1, 1) at loci 2, 3 and 5 respectively. Parameter *haploFreq* accept one or more lists of loci specifying one or more haplotype sites (e.g. `haploFreq=[(0,1,2), (2,3)]` specifies two haplotype sites). The results are saved to dictionaries (with haplotype site as keys) of default dictionaries (with haplotype as keys). For

example, `haploFreq[(0,1,2)][(0,1,1)]` will be the frequency of haplotype (0, 1, 1) at loci (0, 1, 2). Example `statHaploFreq` prints the numbers of genotypes and haplotypes at loci 0, 1 and 2 of a small population. Note that the `viewVars` function defined in module `simuUtil` can make use of a `wxPython` window to view all variables if it is called in GUI mode.

Example: *Counting haplotypes in a population*

```
>>> import simuPOP as sim
>>> from simuPOP.utils import viewVars
>>> pop = sim.Population(100, loci=3)
>>> sim.initGenotype(pop, freq=[0.2, 0.4, 0.4], loci=0)
>>> sim.initGenotype(pop, freq=[0.2, 0.8], loci=2)
>>> sim.stat(pop, genoFreq=[0, 1, 2], haploFreq=[0, 1, 2],
...         vars=['genoNum', 'haploFreq'])
>>> viewVars(pop.vars(), gui=False)
{'genoNum': {0: {(0, 0): 3.0,
                 (0, 1): 7.0,
                 (0, 2): 5.0,
                 (1, 0): 9.0,
                 (1, 1): 14.0,
                 (1, 2): 16.0,
                 (2, 0): 8.0,
                 (2, 1): 14.0,
                 (2, 2): 24.0},
              1: defaultdict({(0, 0): 100.0}),
              2: {(0, 0): 4.0,
                  (0, 1): 19.0,
                  (1, 0): 15.0,
                  (1, 1): 62.0}},
 'haploFreq': {(0, 1, 2): {(0, 0, 0): 0.03,
                           (0, 0, 1): 0.145,
                           (1, 0, 0): 0.055,
                           (1, 0, 1): 0.315,
                           (2, 0, 0): 0.125,
                           (2, 0, 1): 0.33}}}}
```

now exiting runScriptInteractively...

[Download statHaploFreq.py](#)

Note: *haploFreq* does not check if loci in a haplotype site belong to the same chromosome, or if loci are duplicated or in order. It faithfully assemble alleles at specified loci as haplotypes although these haplotypes might not be biologically meaningful.

Note: Counting a large number of haplotypes on long haplotype sites may exhaust the RAM of your computer.

5.11.10 Summary statistics of information fields

Parameter `sumOfInfo`, `meanOfInfo`, `varOfInfo`, `maxOfInfo` and `minOfInfo` are used to calculate the sum, mean, sample variance (), max and min of specified information fields of individuals in all or specified (virtual) subpopulations. The results are saved in dictionaries `sumOfInfo`, `meanOfInfo`, `varOfInfo`, `maxOfInfo` and `minOfInfo` with information fields as keys. For example, parameter `meanOfInfo='age'` calculates the mean age of all individuals and set variable `meanOfInfo['age']`.

Example *statInfo* demonstrates a mixing process of two populations. The population starts with two types of individuals with ancestry values 0 or 1 (information field *anc*). During the evolution, parents mate randomly and the ancestry of offspring is the mean of parental ancestry values. A *Stat* operator is used to calculate the mean and variance of individual ancestry values, and the number of individuals in five ancestry groups. It is not surprising that whereas population mean ancestry does not change, more and more people have about the same number of ancestors from each group and have an ancestry value around 0.5. The variance of ancestry values therefore decreases gradually.

Example: Calculate summary statistics of information fields

```
>>> import simuPOP as sim
>>> import random
>>> pop = sim.Population([500], infoFields='anc')
>>> # Defines VSP 0, 1, 2, 3, 4 by anc.
>>> pop.setVirtualSplitter(sim.InfoSplitter('anc', cutoff=[0.2, 0.4, 0.6, 0.8]))
>>> #
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         # anc is 0 or 1
...         sim.InitInfo(lambda : random.randint(0, 1), infoFields='anc')
...     ],
...     matingScheme=sim.RandomMating(ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.InheritTagger(mode=sim.MEAN, infoFields='anc')
...     ]),
...     postOps=[
...         sim.Stat(popSize=True, meanOfInfo='anc', varOfInfo='anc',
...             subPops=[(0, sim.ALL_AVAIL)]),
...         sim.PyEval(r"Anc: %.2f (%.2f), #inds: %s\n" % (
...             meanOfInfo['anc'], varOfInfo['anc'], " + \
...             ", '.join(['%4d' % x for x in subPopSize]))")
...     ],
...     gen = 5,
... )
Anc: 0.51 (0.12), #inds: 118, 0, 251, 0, 131
Anc: 0.51 (0.06), #inds: 27, 121, 190, 137, 25
Anc: 0.52 (0.03), #inds: 14, 143, 138, 181, 24
Anc: 0.52 (0.02), #inds: 4, 85, 267, 137, 7
Anc: 0.52 (0.01), #inds: 0, 40, 385, 75, 0
5
now exiting runScriptInteractively...
```

[Download statInfo.py](#)

5.11.11 Linkage disequilibrium

Parameter *LD* accepts a list of loci-pairs (e.g. *LD*=[(0, 1) , (2, 3)]) with optional primary alleles at two loci (e.g. *LD*=[(0, 1, 0, 0) , (2, 3)]). For each pair of loci, this operator calculates linkage disequilibrium and optional association measures between them.

Assuming that two loci are both diallelic, one with alleles *a* and *b*, and the other with alleles *c* and *d*. If we denote *p_a*, *p_b*, *p_c*, *p_d* as allele and haplotype frequencies for allele *a* and haplotype *ac*, respectively, the linkage disequilibrium measures **with respect to primaries alleles A and B** are

- Basic LD measure :

D ranges from -0.25 to 0.25. The sign depends on the choice of alleles (*A* and *B*) at two loci.

- Lewontin's where
 D' ranges from -1 to 1. The sign depends on the choice of alleles (A and B) at two loci.
- (in Devlin1995)

If one or both loci have more than 2 alleles, or if no primary allele is specified, the LD measures are calculated as follows:

- If primary alleles are specified, all other alleles are considered as minor alleles with combined frequency (e.g.). The same formulas apply which lead to signed and measures.
- If primary alleles are not specified, these LD measures are calculated as the average of the absolute value of diallelic measures of all allele pairs. For example, the multi-allele version of is
where and iterate through all alleles at the two loci. **In the diallelic case, LD measures will be the absolute value of the single measures** because and only differ by signs.

In another word,

- $LD=[loc1, loc2]$ will yield positive and measures.
- $LD=[loc1, loc2, allele1, allele2]$ will yield signed and measures.
- In the diallelic case, both cases yield identical results except for signs of and .
- In the multi-allelic case, the results can be different because $LD=[loc1, loc2, allele1, allele2]$ combines non-primary alleles and gives a single diallelic measure.

Note: A large number of linkage disequilibrium measures have been used in different disciplines but not all of them are well-accepted. Requests of adding a particular LD measure will be considered when a reliable reference is provided.

Association tests between specified loci could also be calculated using a by table of haplotype frequencies. If primary alleles are specified, non-primary alleles are combined to form a 2 by 2 table (). Otherwise, and are respective numbers of alleles at two loci.

- and its -value (variable LD_ChiSq and LD_ChiSq_p , respectively). A one-side test with degrees of freedom will be used.
- Cramer V statistic (variable $CramerV$):

where equals the total number of haplotypes (for autosomes in diploid populations).

This statistic sets variables LD , LD_prime , $R2$, and optionally $ChiSq$, $ChiSq_p$ and $CramerV$. SubPopulation specific variables can be calculated by specifying variables such as LD_sp and $R2_sp$. Example *statLD* demonstrates how to calculate various LD measures and output selected variables. Note that the significant overall LD between two loci is an artifact of population structure because loci are in linkage equilibrium in each subpopulation.

Example: *Linkage disequilibrium measures*

```
>>> import simuPOP as sim
>>> pop = sim.Population([1000]*2, loci=3)
>>> sim.initGenotype(pop, freq=[0.2, 0.8], subPops=0)
>>> sim.initGenotype(pop, freq=[0.8, 0.2], subPops=1)
>>> sim.stat(pop, LD=[[0, 1, 0, 0], [1, 2]],
...         vars=['LD', 'LD_prime', 'R2', 'LD_ChiSq', 'LD_ChiSq_p', 'CramerV',
...             'LD_prime_sp', 'LD_ChiSq_p_sp'])
>>> from pprint import pprint
>>> pprint(pop.vars())
{'CramerV': {0: defdict({1: 0.3355834766347789})},
```

(continues on next page)

(continued from previous page)

```

        1: defdict({2: 0.39144946095755695})),
'LD': {0: defdict({1: 0.08387987499999999}),
      1: defdict({2: 0.09783043749999992})),
'LD_ChiSq': {0: defdict({1: 450.4650791611408}),
            1: defdict({2: 612.9307219358476})),
'LD_ChiSq_p': {0: defdict({1: 0.0}), 1: defdict({2: 0.0})},
'LD_prime': {0: defdict({1: 0.3425347836362625}),
            1: defdict({2: 0.4057999832524774})),
'R2': {0: defdict({1: 0.1126162697902852}),
      1: defdict({2: 0.15323268048396166})),
'subPop': {0: {'LD_ChiSq_p': {0: defdict({1: 0.03843990070970382}),
                          1: defdict({2: 0.5110492462003573})},
              'LD_prime': {0: defdict({1: -0.1766111690962444}),
                          1: defdict({2: 0.016760924318107204})}},
          1: {'LD_ChiSq_p': {0: defdict({1: 0.8024214035646771}),
                          1: defdict({2: 0.11685510935577492})},
              'LD_prime': {0: defdict({1: -0.02259456714902688}),
                          1: defdict({2: 0.035632559660018596})}}}}
now exiting runScriptInteractively...

```

[Download statLD.py](#)

5.11.12 Genetic association

Genetic association refers to association between individual genotype (alleles or genotype) and phenotype (affection status). There are a large number of statistics tests based on different study designs (e.g. case-control, Pedigree, longitudinal) with different covariate variables. Although specialized software applications should be used for sophisticated statistical analysis, simuPOP provides a number of simple genetic association tests for convenience. These tests

- Are single-locus tests that test specified loci separately.
- Are based on individual affection status. Associations between genotype and quantitative traits are currently unsupported.
- Apply to all individuals in specified (virtual) subpopulations. Because a population usually has much more unaffected individuals than affected ones, it is a common practice to draw certain types of samples (e.g. a case-control sample with the same number of cases and controls) before statistical tests are applied.

simuPOP currently supports the following tests:

- **Allele-based Chi-square test:** This is the basic allele-based test that can be applied to diploid as well as haploid populations. Basically, a 2 by contingency table is set up for each locus with being the number of alleles in cases and controls. A test is applied to each locus and set variables `Allele_ChiSq` and `Allele_ChiSq_p` to the statistic and its two-sided value (with degrees freedom). Note that genotype information is not preserved in such a test.
- **Genotype-based Chi-square test:** This is the genotype-based test for diploid populations. Basically, a 2 by contingency table is set up for each locus with being the number of genotype (unordered pairs of alleles) in cases and controls. A test is applied to each locus and set variables `Geno_ChiSq` and `Geno_ChiSq_p` to the statistic and its two-sided value (with degrees freedom). This test is usually applied to diallelic loci with 3 genotypes (*AA*, *Aa* and *aa*) but it can be applied to loci with more than two alleles as well.
- **Genotype-based trend test:** This Cochran-Armitage test can only be applied to diallelic loci in diploid populations. For each locus, a 2 by 3 contingency table is set up with being the number of genotype (*AA*, *Aa* and *aa* with *A* being the wildtype allele) in cases and controls. A Cochran-Armitage trend test is applied to each locus and set variables `Armitage_p` to its two-sided value.

Example *statAssociation* demonstrates how to apply a penetrance model, draw a case-control sample and apply genetic association tests to an evolving population. In this example, a penetrance model is applied to a locus (locus 3). A Python operator is then used to draw a case-control sample from the population and test genetic association at two surrounding loci. Because these two loci are tightly linked to the disease predisposing locus, they are in strong association with the disease initially. However, because of recombination, such association decays with time at rates depending on their genetic distances to the disease predisposing locus.

Example: *Genetic association tests*

```
>>> import simuPOP as sim
>>> from simuPOP.utils import *
>>> from simuPOP.sampling import drawCaseControlSample
>>> def assoTest(pop):
...     'Draw case-control sample and apply association tests'
...     sample = drawCaseControlSample(pop, cases=500, controls=500)
...     sim.stat(sample, association=(0, 2), vars=['Allele_ChiSq_p', 'Geno_ChiSq_p',
↪ 'Armitage_p'])
...     print('Allele test: %.2e, %.2e, Geno test: %.2e, %.2e, Trend test: %.2e, %.2e
↪ ' \
...           % (sample.dvars().Allele_ChiSq_p[0], sample.dvars().Allele_ChiSq_p[2],
...             sample.dvars().Geno_ChiSq_p[0], sample.dvars().Geno_ChiSq_p[2],
...             sample.dvars().Armitage_p[0], sample.dvars().Armitage_p[2]))
...     return True
...
>>> pop = sim.Population(size=100000, loci=3)
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.5, 0.5]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0]*3, subPops=[(0,0)]),
...         sim.InitGenotype(genotype=[1]*3, subPops=[(0,1)]),
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(loci=[0, 1], rates=[0.01,
↪ 0.005])),
...     postOps=[
...         sim.MaPenetrance(loci=1, penetrance=[0.1, 0.2, 0.4]),
...         sim.PyOperator(func=assoTest, step=20),
...     ],
...     gen = 100
... )
Allele test: 0.00e+00, 0.00e+00, Geno test: 0.00e+00, 0.00e+00, Trend test: 0.00e+00,
↪ 0.00e+00
Allele test: 1.14e-13, 4.44e-16, Geno test: 3.09e-13, 2.66e-15, Trend test: 7.66e-14,
↪ 2.22e-16
Allele test: 1.71e-08, 8.55e-15, Geno test: 4.95e-08, 3.45e-13, Trend test: 1.62e-08,
↪ 7.36e-14
Allele test: 8.57e-09, 7.99e-15, Geno test: 3.09e-08, 2.18e-14, Trend test: 7.05e-09,
↪ 2.66e-15
Allele test: 3.12e-06, 9.05e-09, Geno test: 5.95e-06, 8.83e-08, Trend test: 2.12e-06,
↪ 1.26e-08
100

now exiting runScriptInteractively...
```

[Download statAssociation.py](#)

5.11.13 population structure

Parameter `structure` measures the structure of a population using the following statistics:

- The statistic developed by Nei Nei1973. This statistic is equivalent to Wright's fixation index in the diallelic case so it can be considered as the multi-allele and multi-locus extension of Wright's . It assumes known genotype frequency so it can be used to calculate true of a population when all genotype information is available. This statistic sets a dictionary of locus level (variable `g_st`) and a summary statistics for all loci (variable `G_st`).
- Wright's fixation index calculated using an algorithm developed by Weir1984. This statistic considers existing populations as random samples from an infinite pool of populations with the same ancestral population so it is best to be applied to random samples where true genotype frequencies are unknown. This statistic sets dictionaries of locus level , and (variables `f_st`, `f_is` and `f_it`), and summary statistics for all loci (variables `F_st`, `F_is` and `F_it`) . When heterozygote count is unavailable (non-diploid population, loci on sex chromosomes and mitochondrial chromosomes), simuPOP uses expected heterozygosity to estimate this quantity.

These statistics by default uses all existing subpopulations, but it can also be applied to a subset of subpopulations, or even virtual subpopulations using parameter `subPops`. That is to say, you can measure the genetic difference between males and females using `subPops=[(0,0), (0,1)]` if a `SexSplitter` is used to define two virtual subpopulations with male and female individuals respectively.

Example `statStructure` demonstrate a simulation with two replicates. In the first replicate, three subpopulations evolve separately without migration and become more and more genetically distinct. In the second replicate, a low level migration is applied between subpopulations so the population structure is kept at a low level.

Example: *Measure of population structure*

```
>>> import simuPOP as sim
>>> from simuPOP.utils import migrIslandRates
>>> simu = sim.Simulator(sim.Population([5000]*3, loci=10, infoFields='migrate_to'),
...     rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     preOps=sim.Migrator(rate=migrIslandRates(0.01, 3), reps=1),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(structure=range(10), step=40),
...         sim.PyEval("'Fst=%.3f (rep=%d without migration) ' % (F_st, rep)",
↪step=40, reps=0),
...         sim.PyEval("'Fst=%.3f (rep=%d with migration) ' % (F_st, rep)", step=40,
↪reps=1),
...         sim.PyOutput('\n', reps=-1, step=40)
...     ],
...     gen = 200
... )
Fst=0.000 (rep=0 without migration) Fst=0.000 (rep=1 with migration)
Fst=0.003 (rep=0 without migration) Fst=0.002 (rep=1 with migration)
Fst=0.006 (rep=0 without migration) Fst=0.002 (rep=1 with migration)
Fst=0.008 (rep=0 without migration) Fst=0.003 (rep=1 with migration)
Fst=0.010 (rep=0 without migration) Fst=0.001 (rep=1 with migration)
(200, 200)

now exiting runScriptInteractively...
```

[Download statStructure.py](#)

5.11.14 Hardy-Weinberg equilibrium test

Parameter `HWE` accepts a list of loci at which exact Hardy Weinberg equilibrium tests are applied. The p -values of the tests are assigned to a dictionary `HWE`. Example *statHWE* demonstrates how Hardy Weinberg equilibrium is reached in one generation.

Example: *Hardy Weinberg Equilibrium test*

```
>>> import simuPOP as sim
>>> pop = sim.Population([1000], loci=1)
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.4, 0.4, 0.2]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(genotype=[0,0], subPops=[(0,0)]),
...         sim.InitGenotype(genotype=[0,1], subPops=[(0,1)]),
...         sim.InitGenotype(genotype=[1,1], subPops=[(0,2)]),
...     ],
...     preOps=[
...         sim.Stat(HWE=0, genoFreq=0),
...         sim.PyEval(r'"HWE p-value: %.5f (AA: %.2f, Aa: %.2f, aa: %.2f)\n" %_
↳ (HWE[0], '
...             'genoFreq[0][(0,0)], genoFreq[0][(0,1)] + genoFreq[0][(1,0)],_
↳ genoFreq[0][(1,1)])'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(HWE=0, genoFreq=0),
...         sim.PyEval(r'"HWE p-value: %.5f (AA: %.2f, Aa: %.2f, aa: %.2f)\n" %_
↳ (HWE[0], '
...             'genoFreq[0][(0,0)], genoFreq[0][(0,1)] + genoFreq[0][(1,0)],_
↳ genoFreq[0][(1,1)])'),
...     ],
...     gen = 1
... )
HWE p-value: 0.00000 (AA: 0.40, Aa: 0.40, aa: 0.20)
HWE p-value: 0.93636 (AA: 0.38, Aa: 0.48, aa: 0.15)
1

now exiting runScriptInteractively...
```

[Download statHWE.py](#)

5.11.15 Measure of Inbreeding

Inbreeding coefficient at a generation is defined as the probability that the two alleles in a given individual are identical by decent (IBD). Although it is usually very difficult to estimate this quantity, it is easy to observe it directly during evolution if the ancestors of alleles are tracked. This can be done using the lineage module of simuPOP where allelic lineage is tracked during evolution. For example, Example *statIBD* output the frequency of IBD loci in a population of size 500. It also outputs the frequency of IBS (Identical by State), which should always be larger than IBD frequency, and theoretical estimate of the decay of inbreeding coefficient.

Example: *Frequency of IBD as a measure of inbreeding coefficient*

```
>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage')
>>> import simuPOP as sim
```

(continues on next page)

(continued from previous page)

```

>>> pop = sim.Population([500], loci=[1]*100)
>>> pop.evolve(
...     initOps=[
...         sim.InitLineage(),
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2]*5),
...     ],
...     preOps=[
...         sim.Stat(inbreeding=sim.ALL_AVAIL, popSize=True, step=10),
...         sim.PyEval(r'"gen %d: IBD freq %.4f, IBS freq %.4f, est: %.4f\n" % '
...             '(gen, sum(IBD_freq.values()) / len(IBD_freq), '
...             ' sum(IBS_freq.values()) / len(IBS_freq), '
...             ' 1 - (1-1/(2.*popSize))*gen)', step=10)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 100
... )
gen 0: IBD freq 0.0000, IBS freq 0.1994, est: 0.0000
gen 10: IBD freq 0.0084, IBS freq 0.2072, est: 0.0100
gen 20: IBD freq 0.0167, IBS freq 0.2142, est: 0.0198
gen 30: IBD freq 0.0266, IBS freq 0.2204, est: 0.0296
gen 40: IBD freq 0.0380, IBS freq 0.2292, est: 0.0392
gen 50: IBD freq 0.0486, IBS freq 0.2383, est: 0.0488
gen 60: IBD freq 0.0577, IBS freq 0.2457, est: 0.0583
gen 70: IBD freq 0.0689, IBS freq 0.2566, est: 0.0676
gen 80: IBD freq 0.0782, IBS freq 0.2616, est: 0.0769
gen 90: IBD freq 0.0887, IBS freq 0.2638, est: 0.0861
100

now exiting runScriptInteractively...

```

[Download statIBD.py](#)

5.11.16 Effective population size

Effective population size is an important, yet complicated concept in population genetics. Simply put, the effective population size is determined by a mating scheme, namely how parents are selected and how offspring are generated. In the context of forward-time simulation, if we populate an offspring population from a parental population, a true effective population size can be calculated, under certain assumptions, as

where \bar{m} and \bar{v} are the mean and variance of the number of gametes each parent transmits to the offspring generation. Naturally, the number of sex chromosomes transmitted will be different for males and females. This effective size is independent of genotypes and is called the demographic effective size.

Because the calculation of demographic effective size needs to track which alleles are transmitted from parental to offspring population, it has to collect information from both parental and offspring populations, and can only be calculated using the lineage modules of simuPOP. As shown in Example *statNeDemographic*, a *Stat* operator is applied before mating to mark lineage of alleles of each locus with an individual index, and save the IDs of parents in a variable `Ne_demo_base`. After mating, another *Stat* operator is used to count how many alleles each parent has contributed to the offspring generation, and calculate demographic effective size accordingly. This example uses three virtual subpopulations, a whole subpopulation, all male individuals, and all female individuals, and calculated effective size for loci on an autosome, an X chromosome, and a Y chromosome. As we can imagine, the effective size is 0 at the Y chromosome for all females, because no such chromosome is transmitted from the parental population.

Example: *Demographic effective population size*

```

>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage', quiet=True)
>>> import simuPOP as sim
>>> pop = sim.Population([2000], loci=[1]*3,
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=range(3), subPops=[0, (0,0), (0,1)],
...             vars='Ne_demo_base_sp'),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(effectiveSize=range(3), subPops=[0, (0,0), (0,1)],
...             vars='Ne_demo_sp'),
...         sim.PyEval(r'Demographic Ne: %.1f (auto), %.1f (X), %.1f (Y), '
...             r'Males: %.1f, %.1f, %.1f, Females: %.1f, %.1f, %.1f\n'
...             '% tuple([subPop[0]["Ne_demo"][x] for x in (0, 1, 2)] + '
...             '[subPop[(0,0)]["Ne_demo"][x] for x in (0, 1, 2)] + '
...             '[subPop[(0,1)]["Ne_demo"][x] for x in (0, 1, 2)])')
...     ],
...     gen = 5
... )
Demographic Ne: 2021.2 (auto), 1808.8 (X), 1056.1 (Y), Males: 1038.4, 1049.4, 1056.1,
↪ Females: 983.8, 983.8, nan
Demographic Ne: 2024.8 (auto), 1886.4 (X), 918.2 (Y), Males: 965.7, 1014.2, 918.2,
↪ Females: 1063.3, 1063.3, nan
Demographic Ne: 2048.7 (auto), 1858.5 (X), 969.2 (Y), Males: 1023.0, 1037.4, 969.2,
↪ Females: 1025.1, 1025.1, nan
Demographic Ne: 1955.0 (auto), 1790.6 (X), 956.8 (Y), Males: 958.8, 985.2, 956.8,
↪ Females: 996.5, 996.5, nan
Demographic Ne: 2000.5 (auto), 1811.7 (X), 955.1 (Y), Males: 983.8, 966.2, 955.1,
↪ Females: 1016.8, 1016.8, nan
5
now exiting runScriptInteractively...

```

Download statNeDemographic.py

Effective population sizes could also be estimated from genotypes because changes of genotypes reflects properties of the mating scheme. However, it is important to realize that **evolving a population for one generation is only one realization of many possible realizations of the same mating scheme** (effective size). If we consider the demographic effective size as the average effective size of all realizations, estimating effective size from genotypes will be inaccurate unless a large number of unlinked loci are used. The temporal methods essentially try to get better estimate by averaging such realizations across multiple generations, although the demographic effective size might vary due to change of population size.

simuPOP currently provides two temporal methods proposed by Waples (1989) and Jorde & Ryman's (2007). Because these methods estimate effective population size using changes of allele frequencies of samples at two generations, it is necessary to set a baseline generation before any temporal method could be applied.

The baseline information is saved to variable `Ne_temporal_base` when this variable is specified in the `vars` parameter of the `Stat` operator. After the baseline is set, for example, at generation 0, if the operator `Stat` is applied at generations 0, 20, and 40, it will set variable `Ne_waples89_P1`, `Ne_waples89_P2` (for Waples 1989) and

Ne_tempoFS_P1, Ne_tempoFS_P2 (for Jorde & Ryman 2007, as implemented in a package TempoFS) as the census population size at generation 0, estimated effective population sizes between generation 0 and 20 at generation 20, and estimates between 0 and 40 at generation 40. The variables are lists of three elements: the estimated Ne and lower and upper boundaries of the 95% confidence interval.

Sampling plan 1 assumes that samples are drawn with replacement at the first time point so that some of the individuals sampled in the first time period could have contributed genes to subsequent generations (see Nei and Tajima, 1981 Genetics and other papers). simuPOP uses census population (or subpopulation if the statistics are calculated for each subpopulations) size as and consider the sample being a subset of the population (or subpopulation), it should be applied to a virtual subpopulation (e.g. a subset of individuals defined by a *RangeSplitter*) of the whole population. Sample plan 2 treats the sample as a sample from an infinitely-sized population, and should be applied to a population (sample) that is actually extracted from a larger population. Results under both assumptions are calculated and provided so you should choose the ones that match your sampling plan.

Example *statNeTemporal* demonstrates how to calculate temporal effective population sizes at a 20 generation interval during evolution, using a fixed baseline generation at generation 0. The statistics are estimated from genotypes at 50 unlinked loci from 500 random samples from a population of size 2000. Instead of drawing random samples explicitly, this example defines a virtual subpopulation that consists of the first 500 individuals in the population. The Stat operator is applied at generations 0, 20, 40, ..., 100 to this virtual subpopulation, with the first output being the census size (of the sample). Because a standard Wright-Fisher random mating scheme is used, the true effective population size should be around 2000. It would be interesting to adjust this evolutionary process (with population expansion, with varying number of offspring etc) and the method of estimation (sample size, generations between estimates) to see how well this statistic estimate effective population size under different scenarios.

Example: *Temporal effective population size using a fixed baseline sample*

```
>>> import simuPOP as sim
>>> pop = sim.Population([2000], loci=[1]*50)
>>> pop.setVirtualSplitter(sim.RangeSplitter([0, 500]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...                 vars='Ne_temporal_base'),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...                 vars=['Ne_waples89_P1', 'Ne_tempoFS_P1'], step=20),
...         sim.PyEval(r'"Waples Ne: %.1f (%.1f - %.1f), TempoFS: '
...                 r'%.1f (%.1f - %.1f), at generation %d\n" % '
...                 'tuple(Ne_waples89_P1 + Ne_tempoFS_P1 + [gen])', step=20)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 101
... )
Waples Ne: 500.0 (500.0 - 500.0), TempoFS: 500.0 (500.0 - 500.0), at generation 0
Waples Ne: 1853.1 (1155.2 - 3536.1), TempoFS: 1843.2 (1255.1 - 3467.7), at generation_
↪20
Waples Ne: 1537.9 (979.7 - 2452.6), TempoFS: 1565.7 (1117.0 - 2617.2), at generation_
↪40
Waples Ne: 1843.3 (1178.0 - 2872.4), TempoFS: 1963.4 (1332.2 - 3730.9), at generation_
↪60
Waples Ne: 1783.0 (1143.4 - 2710.7), TempoFS: 1807.2 (1291.5 - 3008.7), at generation_
↪80
Waples Ne: 1572.7 (1011.2 - 2346.6), TempoFS: 1639.5 (1205.1 - 2563.6), at generation_
↪100
```

(continues on next page)

(continued from previous page)

```
101
now exiting runScriptInteractively...
```

Download statNeTemporal.py

Instead of using a fixed baseline generation, it is also possible to reset baseline generation during evolution. For example, Example [statNeInterval](#) demonstrates how to calculate temporal effective population sizes at a 20 generation interval during evolution. This example sets variable `Ne_temporal_base` with `Ne_waples89_P1` whenever the Stat operator is applied. This effectively resets the baseline generation to the present generation at generations 0, 20, 40, etc, so baseline generations 0, 20, 40, ... are used at generations 20, 40, This example also demonstrates how to use the suffix parameter to apply the same statistics with different parameters.

Example: *Temporal effective population size between consecutive samples*

```
>>> import simuPOP as sim
>>> pop = sim.Population([2000], loci=[1]*50)
>>> pop.setVirtualSplitter(sim.RangeSplitter([0, 500]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7]),
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...                 vars='Ne_temporal_base'),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)],
...                 vars='Ne_waples89_P1', step=20),
...         sim.Stat(effectiveSize=range(50), subPops=[(0,0)], step=20,
...                 suffix='_i', vars=['Ne_temporal_base', 'Ne_waples89_P1']),
...         sim.PyEval(r'"Waples Ne (till %d): %.1f (%.1f - %.1f), '
...                 r'(interval) %.1f (%.1f - %.1f)\n" % '
...                 'tuple([gen] + Ne_waples89_P1 + Ne_waples89_P1_i)',
...                 step=20)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 101
... )
Waples Ne (till 0): 500.0 (500.0 - 500.0), (interval) 500.0 (500.0 - 500.0)
Waples Ne (till 20): 1853.1 (1155.2 - 3536.1), (interval) 1853.1 (1155.2 - 3536.1)
Waples Ne (till 40): 1537.9 (979.7 - 2452.6), (interval) 2063.7 (1281.1 - 4094.1)
Waples Ne (till 60): 1843.3 (1178.0 - 2872.4), (interval) 1681.9 (1052.1 - 3112.9)
Waples Ne (till 80): 1783.0 (1143.4 - 2710.7), (interval) 1872.7 (1167.0 - 3586.3)
Waples Ne (till 100): 1572.7 (1011.2 - 2346.6), (interval) 2056.1 (1276.6 - 4073.3)
101
now exiting runScriptInteractively...
```

Download statNeInterval.py

Linkage disequilibrium method is another popular method to estimate effective population size. Compared to temporal methods, it has the distinct advantage that it requires only one sample. simuPOP provides a method that is developed by Waples in his 2006 paper. To use this method, you will need to specify variable `Ne_LD` for a random mating scheme, or `Ne_LD_mono` for a monogamous mating scheme. [statNeLD](#) demonstrates this usage. Note that because the LDNe method is sensitive to rare alleles (which can lead to inflated measure of LD), simuPOP provides estimates that ignores alleles with frequencies less than 0 (all alleles are kept), 0.01, 0.02 and 0.05. The results are saved in variable `Ne_LD` as a dictionary with keys 0, 0.01, 0.02, 0.05, and values as lists of estimated effective population sizes

and their 95% confidence intervals. Because of the existence of many rare alleles, the example gives quite different estimates with and without rare alleles (using cutoff=0.02).

Example: *Effective population size estimated using a LD based method*

```
>>> import simuPOP as sim
>>> pop = sim.Population([2000], loci=[1]*50)
>>> pop.setVirtualSplitter(sim.RangeSplitter([0, 500]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.005]*4 + [0.015]*2 + [0.25, 0.7]),
...     ],
...     preOps=[
...         sim.Stat(effectiveSize=sim.ALL_AVAIL, subPops=[(0,0)],
...             vars='Ne_LD', step=20),
...         sim.PyEval(r'"LD Ne (gen %d): %.1f (%.1f - %.1f) '
...             r', %.1f (%.1f - %.1f, adjusted)\n" % '
...             'tuple([gen] + Ne_LD[0.] + Ne_LD[0.02])',
...             step=20)
...     ],
...     matingScheme=sim.RandomMating(),
...     gen = 101
... )
LD Ne (gen 0): 30623.2 (5220.9 - inf), inf (8071.2 - inf, adjusted)
LD Ne (gen 20): 6297.4 (2574.4 - inf), 1900.0 (1160.3 - 4647.8, adjusted)
LD Ne (gen 40): 2187.6 (1554.1 - 3589.2), 2535.5 (1459.2 - 8173.8, adjusted)
LD Ne (gen 60): 2757.8 (1799.2 - 5619.3), 3510.9 (1801.6 - 32066.7, adjusted)
LD Ne (gen 80): 2574.0 (1729.7 - 4828.9), 1813.2 (1197.7 - 3501.7, adjusted)
LD Ne (gen 100): 3234.6 (1819.5 - 12210.9), 2834.8 (1603.4 - 10168.4, adjusted)
101
now exiting runScriptInteractively...
```

[Download statNeLD.py](#)

simuPOP allows you to estimate effective population size using genotypes at selected loci from selected individuals. It is up to you, however, to decide when to apply the operator (pre- or post-mating), how to draw samples, and select the right method for your data. For example, the temporal methods assume discrete generations and no (or slight) selection, migration, and mutation. The LD method assumes that markers are selectively neutral and independent; population has discrete generations and is closed to immigration; and sampling is random. In addition, to keep the interface simple, simuPOP does not provide many options as dedicated programs do (e.g. TempoFS). Please export your samples in other formats (e.g. use operator `Export` (format='GENEPOP') or function `export` (pop, format='GENEPOP')) from module `simuPOP.utiles` and use these programs if you need such flexibilities.

5.11.17 Other statistics

If you need other statistics, a popular approach is to define them using Python operators. If your statistics is based on existing statistics such as allele frequency, it is a good idea to calculate existing statistics using a `stat` function and derive your statistics from population variables. Please refer to the last chapter of this guide on an example.

If you would like to calculate some summary statistics that involves individual information fields but cannot be calculated using parameters such as `minOfInfo`, you can try to use operators such as `InfoExec` to process individuals one by one and collect result. For example, you can use operators

```
PyExec('s=0')
InfoExec('s+=x*x')
PyEval('s')
```

to calculate and report where *x* is an information field during evolution. This makes use of the fact that operator *InfoExec* goes through all individuals and evaluate the statement.

If performance becomes a problem, you might want to have a look at the source code of simuPOP and implement your statistics at the C++ level. If you believe that your statistics are popular enough, please send your implementation to the simuPOP mailinglist for possible inclusion of your statistics into simuPOP.

5.11.18 Support for sex and customized chromosome types

simuPOP supports statistics calculation for loci on sex chromosomes. For example, when pair-wise difference between haplotypes is calculated using parameter *neutrality*, it will pick the right haplotypes for X, and Y chromosomes. However, because *neutrality* is calculated based on a group of haplotypes of all specified loci, even if the loci are collected across chromosomes, you can not use operator

```
Stat(neutrality=ALL_AVAIL)
```

if the loci are selected from chromosomes of different types, because different numbers of haplotypes exists on these chromosomes. To calculate *Pi* for these chromosomes, you would have to calculate them separately, using operators such as

```
Stat(neutrality=range(30,40), suffix='_X')
Stat(neutrality=range(40,50), suffix='_Y')
```

so that all specified loci are on the same type of chromosomes. Here we use parameter *suffix* to avoid conflict of variable names because both operator would produce the same variable *Pi* without this parameter.

The case with customized chromosomes are more complex because the meaning of these chromosomes are defined by users. If these chromosomes are mitochondrial DNAs, only chromosomes from the females are carrying useful information. If you would like to calculate, for example, the *Pi* statistics for these chromosomes, you will have to explicitly selected females for calculation. This can be done by operator

```
Stat(neutrality=range(50,60), vsps=[(ALL_AVAIL, 'FEMALE')], suffix='_mt')
```

if VSPs have been created by a *SexSplitter*.

Example *statChromTypes* demonstrates the use of these operators. This example intentionally initializes all individuals with the same haplotypes on all chromosomes (the *InitGenotype* operator ignores chromosome types). Because of different chromosome types, four *Stat* operators are used to get the *Pi* statistics for them. These operators return different results because different sets of haplotypes are picked for the calculation of this statistics.

Example: *Statistics for sex and customized chromosome types*

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[5]*4,
...     chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y, sim.
↳ MITOCHONDRIAL])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(haplotypes=[ [0, 1, 2, 0, 1]*4, [2, 1, 0, 2, 3]*4 ],
...         prop=[0.4, 0.6])),
```

(continues on next page)

(continued from previous page)

```

...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.MitochondrialGenoTransmitter()]),
...     preOps=[
...         sim.Stat(neutrality=range(5)),
...         sim.Stat(neutrality=range(5, 10), suffix='_X'),
...         sim.Stat(neutrality=range(10, 15), suffix='_Y'),
...         sim.Stat(neutrality=range(15, 20), suffix='_mt'),
...         sim.PyEval(r'("%.3f %.3f %.3f %.3f\n" % (Pi, Pi_X, Pi_Y, Pi_mt)'),
...     ],
...     gen = 2
... )
1.921 1.900 1.973 1.914
1.931 1.921 1.957 1.945
2

now exiting runScriptInteractively...

```

[Download statChromTypes.py](#)

5.12 Conditional operators

5.12.1 Conditional operator (operator `IfElse`) *

Operator `IfElse` provides a simple way to conditionally apply an operator. The condition can be a fixed condition, a expression (a string) that will be evaluated in a population's local namespace or a user-defined function when it is applied to the population.

The first case is used to control the execution of certain operators depending on user input. For example, Example *IfElseFixed* determines whether or not some outputs should be given depending on a variable `verbose`. Note that the applicability of the conditional operators are determined by the `IfElse` operator and individual operators. That is to say, the parameters `begin`, `step`, `end`, `at`, and `reps` of operators in `ifOps` and `elseOps` are only honored when operator `IfElse` is applied.

Example: *A conditional operator with fixed condition*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=1000, loci=1)
>>> verbose = True
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=sim.IfElse(verbose,
...         ifOps=[
...             sim.Stat(alleleFreq=0),
...             sim.PyEval(r'"Gen: %3d, allele freq: %.3f\n" % (gen, alleleFreq[0][1])
... ↪",
...             step=5)
...         ],
...     ],

```

(continues on next page)

(continued from previous page)

```

...         begin=10),
...     gen = 30
... )
Gen: 10, allele freq: 0.483
Gen: 15, allele freq: 0.455
Gen: 20, allele freq: 0.481
Gen: 25, allele freq: 0.481
30

now exiting runScriptInteractively...
```

Download IfElseFixed.py

When a string is specified, it will be considered as an expression and be evaluated in a population's namespace. The return value will be used to determine if an operator should be executed. For example, you can re-introduce a mutant if it gets lost in the population, output a warning when certain condition is met, or record the occurrence of certain events in a population. For example, Example *IfElse* records the number of generations the frequency of an allele goes below 0.4 and beyond 0.6 before it gets lost or fixed in the population. Note that a list of else-operators can also be executed when the condition is not met.

Example: *A conditional operator with dynamic condition*

```

>>> import simuPOP as sim
>>> simu = sim.Simulator(
...     sim.Population(size=1000, loci=1),
...     rep=4)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.PyExec('below40, above60 = 0, 0')
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.IfElse('alleleFreq[0][1] < 0.4',
...             sim.PyExec('below40 += 1')),
...         sim.IfElse('alleleFreq[0][1] > 0.6',
...             sim.PyExec('above60 += 1')),
...         sim.IfElse('len(alleleFreq[0]) == 1',
...             sim.PyExec('stoppedAt = gen')),
...         sim.TerminateIf('len(alleleFreq[0]) == 1')
...     ]
... )
(892, 1898, 4001, 2946)
>>> for pop in simu.populations():
...     print('Overall: %4d, below 40%: %4d, above 60%: %4d' % \
...         (pop.dvars().stoppedAt, pop.dvars().below40, pop.dvars().above60))
...
Overall: 891, below 40%: 20, above 60%: 515
Overall: 1897, below 40%: 1039, above 60%: 51
Overall: 4000, below 40%: 2878, above 60%: 0
Overall: 2945, below 40%: 198, above 60%: 1731

now exiting runScriptInteractively...
```

Download IfElse.py

In the last case, a user-defined function can be specified. This function should accept parameter `pop` when the operator is applied to a population, and one or more parameters `pop`, `off`, `dad` and `mom` when it is applied during-mating. The later could be used to apply different during-mating operators for different types of parents or offspring. For example, Example *pedigreeMatingAgeStructured* in Chapter 6 uses a *CloneGenoTransmitter* when only one parent is available (when parameter `mom` is `None`), and a *MendelianGenoTransmitter* when two parents are available.

5.12.2 Conditionally terminate an evolutionary process (operator `TerminateIf`)

Operator *TerminateIf* has been described and used in several examples such as Example *simuGen*, *expression* and *IfElse*. This operator accept an Python expression and terminate the evolution of the population being applied if the expression is evaluated to be `True`. This operator is well suited for situations where the number of generations to evolve cannot be determined in advance.

If a *TerminateIf* operator is applied to the offspring generation, the evolutionary cycle is considered to be completed. If the evolution is terminated before mating, the evolutionary cycle is condered to be incomplete. Such a difference can be important if the number of generations that have been involved is important for your analysis.

A less-known feature of operator *TerminateIf* is its ability to terminate the evolution of all replicates, using parameter `stopAll=True`. For example, Example *Terminatelf* terminates the evolution of all populations when one of the populations gets fixed. The return value of `simu.evolve` shows that some populations have evolved one generation less than the population being fixed.

Example: *Terminate the evolution of all populations in a simulator*

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(
...     sim.Population(size=100, loci=1),
...     rep=10)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...         sim.TerminateIf('len(alleleFreq[0]) == 1', stopAll=True)
...     ]
... )
(88, 88, 88, 88, 87, 87, 87, 87, 87, 87)
>>>

now exiting runScriptInteractively...
```

[Download TerminateIf.py](#)

5.12.3 Conditional removal of individuals (operator `DiscardIf`)

Operator *DiscardIf* accepts a fixed condition or probability, or a condition or a Python function that returns either `True/False` or a probability to remove an individual. When it is applied during mating, it will evaluate the condition or call the function for each offspring, and discard the offspring if the return value of the expression or function is `True`, or remove at a probability if the return value is a number between 0 and 1. The python expression accepts information fields as variables so operator *DiscardIf*('age > 80') will discard all individuals with age > 80, and *DiscardIf*('1-fitness') will remove individuals according to 1 minus their fitness. Optionally, the offspring itself can be used in the expression if parameter `exposeInd` is used to set the variable name of the offspring.

Alternatively, a Python function can be passed to this operator. This function should be defined with parameters `pop`, `off`, `mom`, `dad` or names of information fields. For example, `DiscardIf(lambda age: age > 80)` will remove individuals with `age > 80`.

A constant expression is also allowed in this operator. A fixed condition or number is acceptable so `DiscardIf(0.1)` will randomly remove 10% of all individuals. Although it does not make sense to use `DiscardIf(True)` because all offspring will be discarded, it is quite useful to use this operator in the context of `DiscardIf(True, subPops=[(0, 0)])` to remove all individuals in a virtual subpopulation. If virtual subpopulation `(0, 0)` is defined as all individuals with `age > 80`, the last method achieves the same effect as the first two methods.

Example `DiscardIf` demonstrates an interesting application of this operator. This example evolves a population for one generation. Instead of keeping all offspring, it keeps only 500 affected and 500 unaffected offspring. This is achieved by defining virtual subpopulations by affection status and range, and discard the first 500 offspring if they are unaffected, and the last 500 offspring if they are affected.

Example: Use operator `DiscardIf` to generate case control samples

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=500, loci=1)
>>> pop.setVirtualSplitter(sim.ProductSplitter([
...     sim.AffectionSplitter(),
...     sim.RangeSplitter([[0, 500], [500, 1000]]),
... ])
... )
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.MaPenetrance(loci=0, penetrance=[0, 0.01, 0.1]),
...             sim.DiscardIf(True, subPops=[
...                 (0, 'Unaffected, Range [0, 500)'),
...                 (0, 'Affected, Range [500, 1000)'))
...         ],
...         subPopSize=1000,
...     ),
...     gen = 1
... )
1
>>> sim.stat(pop, numOfAffected=True)
>>> print(pop.dvars().numOfAffected, pop.dvars().numOfUnaffected)
500 500

now exiting runScriptInteractively...
```

[Download DiscardIf.py](#)

5.13 Miscellaneous operators

5.13.1 An operator that does nothing (operator `NoneOp`)

Operator `NoneOp` does nothing when it is applied to a population. It provides a placeholder when an operator is needed but no action is required. Example `NoneOp` demonstrates a typical usage of this operator

```

if hasSelection:
    sel = MapSelector(loci=[0], fitness=[1, 0.99, 0.98])
else:
    sel = NoneOp()
#
simu.evolve(
    preOps=[sel], # and other operators
    matingScheme=RandomMating(),
    gen=10
)

```

5.13.2 dump the content of a population (operator *Dumper*)

Operator *Dumper* and its function form *dump* has been used extensively in this guide. They are perfect for demonstration and debugging purposes because they display all properties of a population in a human readable format. They are, however, rarely used in realistic settings because outputting a large population to your terminal can be disastrous.

Even with modestly-sized populations, it is a good idea to dump only parts of the population that you are interested. For example, you can use parameter `genotype=False` to stop outputting individual genotype, `structure=False` to stop outputting genotypic and population structure information, `loci=range(5)` to output genotype only at the first five loci, `max=N` to output only the first N individuals (default to 100), `subPops=[(0, 0)]` to output, for example, only the first virtual subpopulation in subpopulation 0. Multiple virtual subpopulations are allowed and you can even use `subPops=[(ALL_AVAIL, 0)]` to go through a specific virtual subpopulation of all subpopulations. This operator by default only dump the present generation but you can set `ancGens` to a list of generation numbers or `ALL_AVAIL` to dump part or all ancestral generations. Finally, if there are more than 10 alleles, you can set the width at which each allele will be printed. The following example (Example *Dumper*) presents a rather complicated usage of this operator.

Example: *dump the content of a population*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[10, 10], loci=[20, 30], infoFields='gen',
...     ancGen=-1)
>>> sim.initSex(pop)
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop1 = pop.clone()
>>> sim.initGenotype(pop, freq=[0]*20 + [0.1]*10)
>>> pop.setIndInfo(1, 'gen')
>>> sim.initGenotype(pop1, freq=[0]*50 + [0.1]*10)
>>> pop1.setIndInfo(2, 'gen')
>>> pop.push(pop1)
>>> sim.dump(pop, width=3, loci=[5, 6, 30], subPops=([0, 0], [1, 1]),
...     max=10, structure=False)
SubPopulation 0,0 (Male), 5 Individuals:
  2: MU  56 54 52 |  58 54 51 |  2
  3: MU  52 50 51 |  56 51 50 |  2
  4: MU  50 53 52 |  52 59 56 |  2
  5: MU  57 54 56 |  57 57 53 |  2
  6: MU  59 54 54 |  57 51 50 |  2
SubPopulation 1,1 (Female), 7 Individuals:
 10: FU  54 53 57 |  59 59 59 |  2
 11: FU  55 59 51 |  59 51 58 |  2
 12: FU  55 58 58 |  57 54 58 |  2
 14: FU  53 57 52 |  51 54 58 |  2
 15: FU  51 58 59 |  54 52 54 |  2

```

(continues on next page)

(continued from previous page)

```

>>> # list all male individuals in all subpopulations
>>> sim.dump(pop, width=3, loci=[5, 6, 30], subPops=[(sim.ALL_AVAIL, 0)],
...         max=10, structure=False)
SubPopulation 0,0 (Male), 5 Individuals:
  2: MU  56 54 52 |  58 54 51 |  2
  3: MU  52 50 51 |  56 51 50 |  2
  4: MU  50 53 52 |  52 59 56 |  2
  5: MU  57 54 56 |  57 57 53 |  2
  6: MU  59 54 54 |  57 51 50 |  2
SubPopulation 1,0 (Male), 3 Individuals:
 13: MU  55 52 53 |  57 56 52 |  2
 17: MU  55 51 51 |  57 55 51 |  2
 19: MU  56 54 53 |  58 58 56 |  2

now exiting runScriptInteractively...

```

[Download Dumper.py](#)

5.13.3 Save a population during evolution (operator *SavePopulation*)

Because it is usually not feasible to store all parental generations of an evolving population, it is a common practise to save snapshots of a population during an evolutionary process for further analysis. Operator *SavePopulation* is designed for this purpose. When it is applied to a population, it will save the population to a file specified by parameter output.

The tricky part is that populations at different generations need to be saved to different filenames so the expression version of parameter output needs to be used (see operator *BaseOperator* for details). For example, expression 'snapshot_%d_%d.pop' % (rep, gen) is used in Example *SavePopulation* to save population to files such as snapshot_5_20.pop during the evolution.

Example: *Save snapshots of an evolving population*

```

>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100, loci=2),
...     rep=5)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.8])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=sim.SavePopulation(output="!'snapshot_%d_%d.pop' % (rep, gen)",
...         step = 10),
...     gen = 50
... )
(50, 50, 50, 50, 50)

now exiting runScriptInteractively...

```

[Download SavePopulation.py](#)

5.13.4 Pause and resume an evolutionary process (operator `Pause`) *

If you are presenting an evolutionary process in public, you might want to temporarily stop the evolution so that your audience can have a better look at intermediate results or figures. If you have an exceptionally long evolutionary process, you might want to examine the status of the evolution process from time to time. These can be done using a `Pause` operator.

The `Pause` operator can stop the evolution at specified generations, or when you press a key. In the first case, you usually specify the generations to `Pause` (e.g. `Pause(step=1000)`) so that you can examine the status of a simulation from time to time. In the second case, you can apply the operator at each generation and `Pause` the simulation when you press a key (e.g. `Pause(stopOnKeyStroke=True)`). A specific key can be specified so that you can use different keys to stop different populations, as shown in Example [Pause](#).

Example: *Pause the evolution of a simulation*

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(100), rep=10)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[sim.Pause(stopOnKeyStroke=str(x), reps=x) for x in range(10)],
...     gen = 100
... )
(100, 100, 100, 100, 100, 100, 100, 100, 100, 100)

now exiting runScriptInteractively...
```

[Download Pause.py](#)

When a simulation is `Paused`, you are given the options to resume evolution, stop the evolution of the `Paused` population or all populations, or enter an interactive Python shell to examine the status of a population, which will be available in the Python shell as `pop_X_Y` where `X` and `Y` are generation and replicate number of the population, respectively. The evolution will resume after you exit the Python shell.

5.13.5 Measuring execution time of operators (operator `TicToc`) *

The `TicToc` operator can be used to measure the time between two events during an evolutionary process. It outputs the elapsed time since the last time it is called, and the overall time since the operator is created. It is very flexible in that you can measure the time spent for mating in an evolutionary cycle if you apply it before and after mating, and you can measure time spent for several evolutionary cycles using generation applicability parameters such as `step` and `at`. The latter usage is demonstrated in Example [TicToc](#).

Example: *Monitor the performance of operators*

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(10000, loci=[100]*5), rep=2)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.1, 0.9])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=0),
...     ]
... )
```

(continues on next page)

(continued from previous page)

```

...         sim.TicToc(step=50, reps=-1),
...     ],
...     gen = 101
... )
Start stopwatch.
Elapsed time: 5.00s   Overall time: 5.00s
Elapsed time: 4.00s   Overall time: 9.00s
(101, 101)

now exiting runScriptInteractively...
```

[Download TicToc.py](#)

5.14 Hybrid and Python operators

5.14.1 Hybrid operators

Despite the large number of built-in operators, it is obviously not possible to implement every genetics models available. For example, although simuPOP provides several penetrance models, a user may want to try a customized one. In this case, one can use a *hybrid operator*.

A *hybrid operator* is an operator that calls a user-defined function when its applied to a population. The number and meaning of input parameters and return values vary from operator to operator. For example, a hybrid mutator sends a to-be-mutated allele to a user-defined function and use its return value as a mutant allele. A hybrid selector uses the return value of a user defined function as individual fitness. Such an operator handles the routine part of the work (e.g. scan through a chromosome and determine which allele needs to be mutated), and leave the creative part to users. Such a mutator can be used to implement complicated genetic models such as an asymmetric stepwise mutation model for microsatellite markers.

simuPOP operators use parameter names to determine which information should be passed to a user-defined function. For example, a hybrid quantitative trait operator recognizes parameters `ind`, `geno`, `gen` and names of information fields such as `smoking`. If your model depends on genotype, you could provide a function with parameter `geno` (e.g. `func(geno)`); if your model depends on smoking and genotype, you could provide a function with parameters `geno` and `smoking` (e.g. `func(geno, smoking)`); if you model depends on individual sex, you can use a function that passes the whole individual (e.g. `func(ind)`) so that you could check individual sex. When a hybrid operator is applied to a population, it will check the parameter names of provided Python function and send requested information automatically.

For example, Example [hybridOperator](#) defines a three- locus heterogeneity penetrance model Risch1990 that yields positive penetrance only when at least two disease susceptibility alleles are available. The underlying mechanism of this operator is that for each individual, simuPOP will collect genotype at specified loci (parameter `loci`) and send them to function `myPenetrance` and evaluate. The return values are used as the penetrance value of the individual, which is then interpreted as the probability that this individual will become affected.

Example: *Use a hybrid operator*

```

>>> import simuPOP as sim
>>> def myPenetrance(geno):
...     'A three-locus heterogeneity penetrance model'
...     if sum(geno) < 2:
...         return 0
...     else:
...         return sum(geno)*0.1
... 
```

(continues on next page)

(continued from previous page)

```

>>> pop = sim.Population(1000, loci=[20]*3)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.8, 0.2])
...     ],
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.PyPenetrance(func=myPenetrance, loci=[10, 30, 50]),
...         sim.Stat(numOfAffected=True),
...         sim.PyEval(r"%d: %d\n" % (gen, numOfAffected))
...     ],
...     gen = 5
... )
0: 97
1: 96
2: 78
3: 95
4: 80
5
now exiting runScriptInteractively...

```

[Download hybrid.py](#)

5.14.2 Python operator `PyOperator` *

If hybrid operators are still not flexible enough, you can always resort to a pure-Python operator `PyOperator`. This operator has full access to the evolving population (or parents and offspring when applied during-mating), and can therefore perform arbitrary operations.

A `PyOperator` that is applied pre- or post- mating expects a function with one or both parameters `pop` and `param`, where `pop` is the population being applied, and `param` is optional, depending on whether or not a parameter is passed to the `PyOperator()` constructor. Function `func` can perform arbitrary action to `pop` and must return `True` or `False`. **The evolution of `pop` will be stopped if this function returns `False`.** This is essentially how operator `TerminateIf` works. Alternatively, this callback function can accept `ind` as one of the parameters. In this case, the function will be called for all individuals or individuals in specified (virtual) subpopulations. **Individuals will be removed from the population if this function returns `False`.**

Example `PyOperator` defines such a function. It accepts a cutoff value and two mutation rates as parameters. It then calculate the frequency of allele 1 at each locus and apply a two-allele model at high mutation rate if the frequency is lower than the cutoff and a low mutation rate otherwise. The `kAlleleMutate` function is the function form of a mutator `KAlleleMutator` (see Section [subsec_Function_form](#) for details).

Example: *A frequency dependent mutation operator*

```

import simuPOP as sim
def dynaMutator(pop, param):
    '''This mutator mutates common loci with low mutation rate and rare
    loci with high mutation rate, as an attempt to raise allele frequency
    of rare loci to an higher level.'''
    # unpack parameter
    (cutoff, mu1, mu2) = param;
    sim.stat(pop, alleleFreq=range(pop.totNumLoci()))
    for i in range(pop.totNumLoci()):

```

(continues on next page)

(continued from previous page)

```

# Get the frequency of allele 1 (disease allele)
if pop.dvars().alleleFreq[i][1] < cutoff:
    sim.kAlleleMutate(pop, k=2, rates=mu1, loci=[i])
else:
    sim.kAlleleMutate(pop, k=2, rates=mu2, loci=[i])
return True

```

Download PyOperator.py

Example *usePyOperator* demonstrates how to use this operator. It first initializes the population using two *InitGenotype* operators that initialize loci with different allele frequencies. It applies a *PyOperator* with function *dynaMutator* and a tuple of parameters. Allele frequencies at all loci are printed at generation 0, 10, 20, and 30. Note that this *PyOperator* is applied at to the parental generation so allele frequencies have to be recalculated to be used by post- mating operator *PyEval*.

Example: *Use a PyOperator during evolution*

```

>>> pop = sim.Population(size=10000, loci=[2, 3])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.99, .01], loci=[0, 2, 4]),
...         sim.InitGenotype(freq=[.8, .2], loci=[1, 3])
...     ],
...     preOps=sim.PyOperator(func=dynaMutator, param=(.2, 1e-2, 1e-5)),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=range(5), step=10),
...         sim.PyEval(r"' '.join(['%.2f' % alleleFreq[x][1] for x in range(5)]) + '\n'
...     ],
...     step=10),
...     ],
...     gen = 31
... )
0.02 0.20 0.02 0.20 0.02
0.11 0.22 0.11 0.20 0.11
0.19 0.21 0.20 0.20 0.18
0.21 0.21 0.22 0.21 0.21
31

now exiting runScriptInteractively...

```

Download PyOperator.py

5.14.3 During-mating Python operator *

A *PyOperator* can also be applied during-mating. They can be used to filter out unwanted offspring (by returning *False* in a user-defined function), modify offspring, calculate statistics, or pass additional information from parents to offspring. Depending the names of parameters of your function, the Python operator will pass offspring (parameter *off*), his or her parents (parameter *dad* and *mom*), the whole population (parameter *pop*) and an optional parameter (parameter *param*) to this function. For example, function *func(off)* will accept references to an offspring, and *func(off, mom, dad)* will accept references to both offspring and his or her parents.

Example *duringMatingPyOperator* demonstrates the use of a during-mating Python operator. This operator rejects an offspring if it has allele 1 at the first locus of the first homologous chromosome, and results in an offspring population without such individuals.

Example: *Use a during-mating PyOperator*

```
>>> import simuPOP as sim
>>> def rejectInd(off):
...     'reject an individual if it off.allele(0) == 1'
...     return off.allele(0) == 0
...
>>> pop = sim.Population(size=100, loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.PyOperator(func=rejectInd)
...         ]
...     ),
...     gen = 1
... )
1
>>> # You should see no individual with allele 1 at locus 0, ploidy 0.
>>> pop.genotype()[ :20]
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

now exiting runScriptInteractively...
```

Download [pyDuringMatingOperator.py](#)

PyOperator is the most powerful operator in simuPOP and has been widely used, for example, to calculate statistics and is not supported by the *Stat()* operator, to examine population property during evolution, or prepare populations for a special mating scheme. However, because *PyOperator* works in the Python interpreter, it is expected that it runs slower than operators that are implemented at the C/C++ level. If performance becomes an issue, you can re-implement part or all the operator in C++. Section [subsec_Using_C++](#) describes how to do this.

5.14.4 Define your own operators *

PyOperator is a Python class so you can derive your own operator from this operator. The tricky part is that the constructor of the derived operator needs to call the `__init__` function of *PyOperator* will proper functions. This technique has been used by simuPOP in a number of occasions. For example, the *VarPlotter* operator defined in `plotter.py` is derived from *PyOperator*. This class encapsulates several different plot class that uses `rpy` to plot python expressions. One of the plotters is passed to the `func` parameter of *PyOperator*. `__init__` so that it can be called when this operator is applied.

Example *sequentialSelfing* rewrites the *dynaMutator* defined in Example *PyOperator* into a derived operator. The parameters are now passed to the constructor of *dynaMutator* and are saved as member variables. A member function `mutate` is defined and is passed to the constructor of *PyOperator*. Other than making *dynaMutator* look like a real simuPOP operator, this example does not show a lot of advantage over defining a function. However, when the operator gets complicated (as in the case for *VarPlotter*), the object oriented implementation will prevail.

Example: *Define a new Python operator*

```
>>> import simuPOP as sim
>>> class dynaMutator(sim.PyOperator):
...     '''This mutator mutates commom loci with low mutation rate and rare
...     loci with high mutation rate, as an attempt to raise allele frequency
...     of rare loci to an higher level.'''
```

(continues on next page)

(continued from previous page)

```

...     def __init__(self, cutoff, mu1, mu2, *args, **kwargs):
...         self.cutoff = cutoff
...         self.mu1 = mu1
...         self.mu2 = mu2
...         sim.PyOperator.__init__(self, func=self.mutate, *args, **kwargs)
...     #
...     def mutate(self, pop):
...         sim.stat(pop, alleleFreq=range(pop.totNumLoci()))
...         for i in range(pop.totNumLoci()):
...             # Get the frequency of allele 1 (disease allele)
...             if pop.dvars().alleleFreq[i][1] < self.cutoff:
...                 sim.kAlleleMutate(pop, k=2, rates=self.mu1, loci=[i])
...             else:
...                 sim.kAlleleMutate(pop, k=2, rates=self.mu2, loci=[i])
...         return True
...
>>> pop = sim.Population(size=10000, loci=[2, 3])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[.99, .01], loci=[0, 2, 4]),
...         sim.InitGenotype(freq=[.8, .2], loci=[1, 3])
...     ],
...     preOps=dynaMutator(cutoff=.2, mu1=1e-2, mu2=1e-5),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(alleleFreq=range(5), step=10),
...         sim.PyEval(r"' '.join(['%.2f' % alleleFreq[x][1] for x in range(5)]) + '\n'
↪         ",
...         step=10),
...     ],
...     gen = 31
... )
0.02 0.20 0.02 0.20 0.02
0.11 0.22 0.11 0.20 0.11
0.19 0.21 0.20 0.20 0.18
0.21 0.21 0.22 0.21 0.21
31
now exiting runScriptInteractively...

```

[Download newOperator.py](#)

New during-mating operators can be defined similarly. They are usually used to define customized genotype transmitters. Section `subsec_Customized_genotype_transmitter` will describe this feature in detail.

6.1 Mating Schemes

Mating schemes are responsible for populating an offspring generation from the parental generation. There are currently two types of mating schemes

- A **homogeneous mating scheme** is the most flexible and most frequently used mating scheme and is the center topic of this section. A homogeneous mating is composed of a *parent chooser* that is responsible for choosing parent(s) from a (virtual) subpopulation and an *offspring generator* that is used to populate all or part of the offspring generation. During-mating operators are used to transmit genotypes from parents to offspring. Figure [fig_homogeneous_mating_scheme](#) demonstrates this process.
- A **heterogeneous mating scheme** applies several homogeneous mating scheme to different (virtual) subpopulations. Because the division of virtual subpopulations can be arbitrary, this mating scheme can be used to simulate mating in heterogeneous populations such as populations with age structure.
- A **pedigree mating scheme** evolves a population by following the pedigree structure of a pedigree. This mating scheme is used to replay a recorded or manually created evolutionary process.

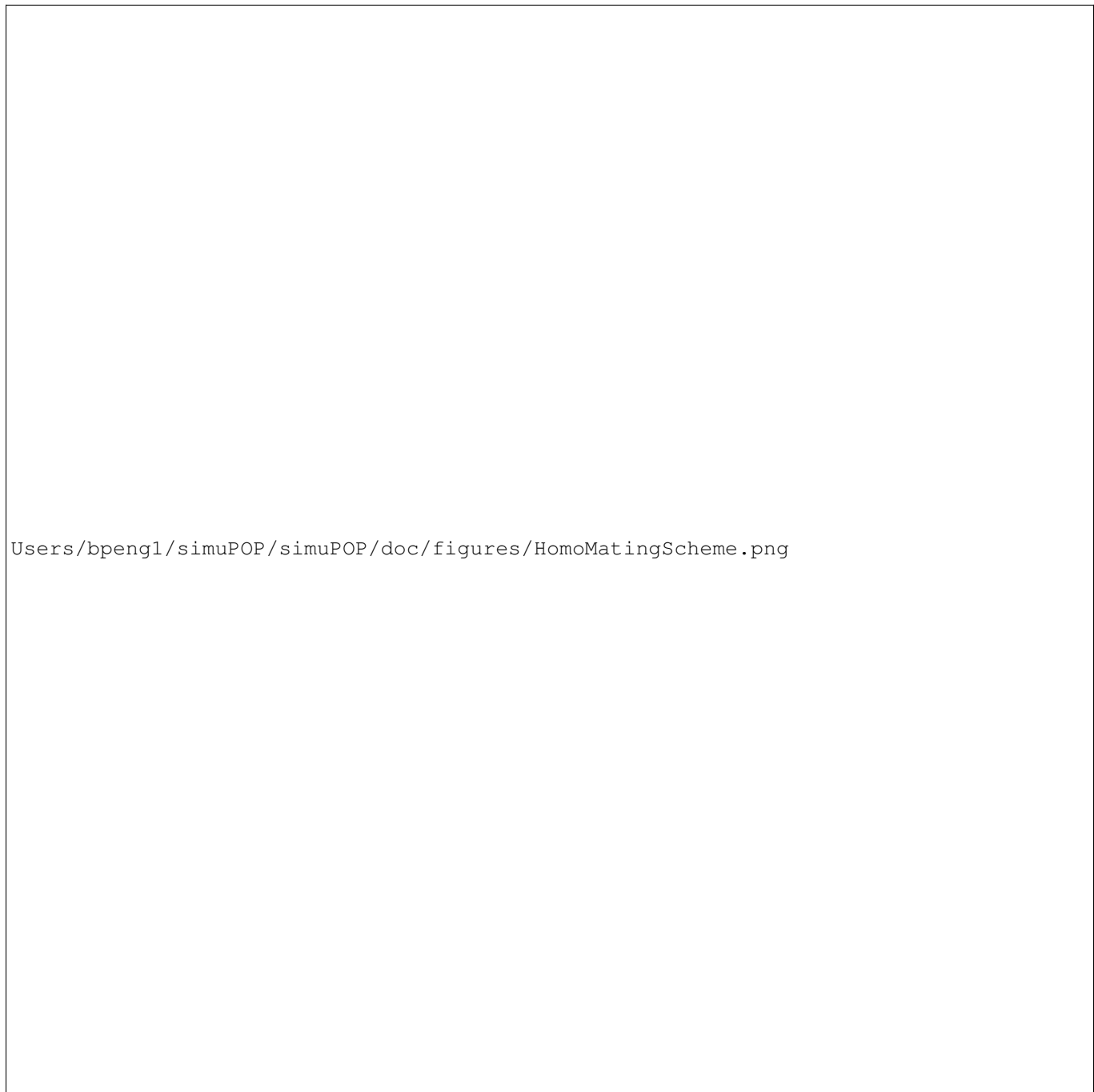
This section describes some standard features of mating schemes and most pre-defined mating schemes. The next section will demonstrate how to build complex nonrandom mating schemes from scratch.

Figure: *A homogeneous mating scheme*

A homogeneous mating scheme is responsible to choose parent(s) from a subpopulation or a virtual subpopulation, and population part or all of the corresponding offspring subpopulation. A parent chooser is used to choose one or two parents from the parental generation, and pass it to an offspring generator, which produces one or more offspring. During mating operators such as taggers and Recombinator can be applied when offspring is generated.

6.1.1 Control the size of the offspring generation

A mating scheme goes through each subpopulation and populates the subpopulations of an offspring generation sequentially. The number of offspring in each subpopulation is determined by the mating scheme, following the following rules:



Users/bpeng1/simuPOP/simuPOP/doc/figures/HomoMatingScheme.png

- A simuPOP mating scheme, by default, produces an offspring generation that has the same subpopulation sizes as the parental generation. This does not guarantee a constant population size because some operators, such as *Migrator* and *DiscardIf* can change population or subpopulation sizes.
- If fixed subpopulation sizes are given to parameter `subPopSize`. A mating scheme will generate an offspring generation with specified sizes even if an operator has changed parental population sizes.
- A **demographic function** can be specified to parameter `subPopSize`. This function should take one of the two forms `func(gen)` or `func(gen, pop)` where `gen` is the current generation number and `pop` is the parental population just before mating. This function should return an array of new subpopulation sizes. A single number can be returned if there is only one subpopulation. The *simuPOP.demography* module provides a number of demography-related functions for complex evolutionary scenarios. **Please consider contributing to this module if you have implemented demographic models for particular populations.**

The following examples demonstrate these cases. Example *migrSize* uses a default *RandomMating()* scheme that keeps parental subpopulation sizes. Because migration between two subpopulations are asymmetric, the size of the first subpopulation increases at each generation, although the overall population size keeps constant.

Example: *Free change of subpopulation sizes*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...     matingScheme=sim.RandomMating(),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'%s\n' % subPopSize')
...     ],
...     gen = 3
... )
[843, 657]
[948, 552]
[1010, 490]
3
now exiting runScriptInteractively...
```

[Download migrSize.py](#)

Example *migrFixedSize* uses the same *Migrator* to move individuals between two subpopulations. Because a constant subpopulation size is specified, the offspring generation always has 500 and 1000 individuals in its two subpopulations. Note that operators *Stat* and *PyEval* are applied both before and after mating. It is clear that subpopulation sizes changes before mating as a result of migration, although the pre-mating population sizes vary because of uncertainties of migration.

Example: *Force constant subpopulation sizes*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=[
...         sim.Migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'%s\n' % subPopSize')
...     ],
...     matingScheme=sim.RandomMating(subPopSize=[500, 1000]),
```

(continues on next page)

(continued from previous page)

```

...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[843, 657]
[500, 1000]
[795, 705]
[500, 1000]
[821, 679]
[500, 1000]
3
now exiting runScriptInteractively...

```

[Download migrFixedSize.py](#)

Example *demoFunc* uses a demographic function to control the subpopulation size of the offspring generation. This example implements a linear population expansion model but arbitrarily complex demographic model can be implemented similarly.

Example: *Use a demographic function to control population size*

```

>>> import simuPOP as sim
>>> def demo(gen):
...     return [500 + gen*10, 1000 + gen*10]
...
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.Migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[500, 1000]
[510, 1010]
[520, 1020]
3
now exiting runScriptInteractively...

```

[Download demoFunc.py](#)

If the size of the offspring generation can not be determined directly from generation number, you can pass the parental population as parameter `pop` to the demographic function. For example, Example *demoFunc1* implements a demographic model where a population expand at random numbers at each generation.

Example: *Use parental population to determine the size of offspring population*

```

>>> import simuPOP as sim
>>> import random
>>> def demo(pop):
...     return [x + random.randint(50, 100) for x in pop.subPopSizes()]

```

(continues on next page)

(continued from previous page)

```

...
>>> pop = sim.Population(size=[500, 1000], infoFields='migrate_to')
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[586, 1075]
[649, 1128]
[742, 1214]
3
now exiting runScriptInteractively...

```

Download [demoFunc1.py](#)

In all the above examples, migration and demographic changes are introduced manually to influence the evolution of populations. However, the demographic changes might be driven by other factors such as natural selection so that it is difficult to predict the size of offspring generations in advance. In this case, you can manually remove individuals from parental (or offspring) populations using appropriate operators.

For example, a population in Example [demoBySelection](#) suffers from a sudden reduction of population size (due to perhaps a famine) at generation 3, and a gradual reduction of population size (due to perhaps an outbreak of an infectious disease) after generation 5. The first event is implemented using a [ResizeSubPops](#) operator that directly shrink the population size in half. The second event is implemented using a [MaPenetrance](#) and a [DiscardIf](#) operator. The first operator assigns affection status of each individual using a disease model that involves individual genotype. The second operator discard all individuals that are affected with the disease. Despite of these unfortunate events, the population tries to expand exponentially with offspring population sizes set to 105% of their parental populations.

Example: *Change of population size caused by natural selection*

```

>>> import simuPOP as sim
>>> def demo(pop):
...     return int(pop.popSize() * 1.05)
...
>>> pop = sim.Population(size=10000, loci=1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.7, 0.3])
...     ],
...     preOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%d %s --> " % (gen, subPopSize)'),
...         sim.ResizeSubPops(0, proportions=[0.5], at=2),
...         sim.MaPenetrance(loci=0, penetrance=[0.01, 0.2, 0.6], begin=4),
...         sim.DiscardIf('ind.affected()', exposeInd='ind', begin=4),
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s --> " % subPopSize'),
...     ],
...     matingScheme=sim.RandomMating(subPopSize=demo),
...     postOps=[

```

(continues on next page)

(continued from previous page)

```

...     sim.Stat(popSize=True),
...     sim.PyEval(r'"%s\n" % subPopSize')
... ],
...     gen = 6
... )
0 [10000] --> [10000] --> [10500]
1 [10500] --> [10500] --> [11025]
2 [11025] --> [5512] --> [5787]
3 [5787] --> [5787] --> [6076]
4 [6076] --> [5188] --> [5447]
5 [5447] --> [4845] --> [5087]
6

now exiting runScriptInteractively...
```

Download [demoBySelection.py](#)

6.1.2 Advanced use of demographic functions *

The parental population passed to a demographic function is usually used to determine offspring population size from parental population size. However, because this function is called immediately before mating happens, it provides a good opportunity for you to prepare the parental generation for mating. Such activities could generally be done by operators, but operations related to demographic changes could be done here. For example, Example [advancedDemoFunc](#) uses a demographic function to split populations at certain generation. The advantage of this method over the use of a [SplitSubPops](#) operator (for example as in Example [splitByProp](#)) is that all demographic information presents in the same function so you do not have to worry about changing an operator when your demographic model changes.

Example: *Use a demographic function to split parental population*

```

>>> import simuPOP as sim
>>> def demo(gen, pop):
...     if gen < 2:
...         return 1000 + 100 * gen
...     if gen == 2:
...         # this happens right before mating at generation 2
...         size = pop.popSize()
...         pop.splitSubPop(0, [size // 2, size - size//2])
...         # for generation two and later
...         return [x + 50 * gen for x in pop.subPopSizes()]
...
>>> pop = sim.Population(1000)
>>> pop.evolve(
...     preOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"Gen %d:\t%s (before mating)\t" % (gen, subPopSize)')
...     ],
...     matingScheme=sim.RandomSelection(subPopSize=demo),
...     postOps=[
...         sim.Stat(popSize=True),
...         sim.PyEval(r'"%s (after mating)\n" % subPopSize')
...     ],
...     gen = 5
... )
Gen 0:          [1000] (before mating)  [1000] (after mating)
```

(continues on next page)

(continued from previous page)

```

Gen 1:      [1000] (before mating)  [1100] (after mating)
Gen 2:      [1100] (before mating)  [650, 650] (after mating)
Gen 3:      [650, 650] (before mating)  [800, 800] (after mating)
Gen 4:      [800, 800] (before mating)  [1000, 1000] (after mating)
5
now exiting runScriptInteractively...
```

[Download advancedDemoFunc.py](#)

6.1.3 Determine the number of offspring during mating

simuPOP by default produces only one offspring per mating event. Because more parents are involved in the production of offspring, this setting leads to larger effective population sizes than mating schemes that produce more offspring at each mating event. However, various situations require a larger family size or even varying family sizes. In these cases, parameter `numOffspring` can be used to control the number of offspring that are produced at each mating event. This parameter takes the following types of inputs

- If a single number is given, `numOffspring` offspring are produced at each mating event.
- If a Python function is given, this function will be called each time when a mating event happens. Generation number can be passed to this function as parameter `gen` to allow different numbers of offspring at different generations. A python generator function can also be passed to provide an iterator interface to yield number of offspring for all mating events.
- If a tuple (or list) with more than one numbers is given, the first number must be one of `GEOMETRIC_DISTRIBUTION`, `POISSON_DISTRIBUTION`, `BINOMIAL_DISTRIBUTION` and `UNIFORM_DISTRIBUTION`, with one or two additional parameters.

The number of offspring in the last case will then follow a specific statistical distribution. More specifically,

- `numOffspring=(GEOMETRIC_DISTRIBUTION, p)`: The number of offspring for each mating event follows a geometric distribution with mean and variance :
- `numOffspring=(POISSON_DISTRIBUTION, p)`: The number of offspring for each mating event follows a Poisson distribution with mean and variance . The distribution is

Note that, however, because families with zero offspring are ignored, the distribution of the observed number of offspring (excluding zero) follows a zero-truncated Poisson distribution with probability

The mean number of offspring is therefore , which is 2.31 for .

- `numOffspring=(BINOMIAL_DISTRIBUTION, p, n)` : The number of offspring for each mating event follows a Binomial distribution with mean and variance .

Because families with zero offspring are ignored, the distribution of the observed number of offspring (excluding zero) follows a zero-truncated Binomial distribution, with mean number of offspring being .

- `numOffspring=(UNIFORM_DISTRIBUTION, a, b)` : The number of offspring for each mating event follows a discrete uniform distribution with lower bound and upper bound .

The lower bound of this distribution can be 0 but is identical to the case with .

Example `numOff` demonstrates how to use parameter `numOffspring`. In this example, a function `checkNumOffspring` is defined. It takes a mating scheme as its input parameter and use it to evolve a population with 30 individuals. After evolving a population for one generation, parental indexes are used to identify siblings, and then the number of offspring per mating event.

Example: *Control the number of offspring per mating event.*

```

>>> import simuPOP as sim
>>> def checkNumOffspring(numOffspring, ops=[]):
...     '''Check the number of offspring for each family using
...     information field father_idx
...     '''
...     pop = sim.Population(size=[30], loci=1, infoFields=['father_idx', 'mother_idx',
...     ↪'])
...     pop.evolve(
...         initOps=[
...             sim.InitSex(),
...             sim.InitGenotype(freq=[0.5, 0.5]),
...         ],
...         matingScheme=sim.RandomMating(ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.ParentsTagger(),
...             ] + ops,
...         numOffspring=numOffspring),
...         gen=1)
...     # get the parents of each offspring
...     parents = [(x, y) for x, y in zip(pop.indInfo('mother_idx'),
...     pop.indInfo('father_idx'))]
...     # Individuals with identical parents are considered as siblings.
...     famSize = []
...     lastParent = (-1, -1)
...     for parent in parents:
...         if parent == lastParent:
...             famSize[-1] += 1
...         else:
...             lastParent = parent
...             famSize.append(1)
...     return famSize
>>> # Case 1: produce the given number of offspring
>>> checkNumOffspring(numOffspring=2)
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
>>> # Case 2: Use a Python function
>>> import random
>>> def func(gen):
...     return random.randint(5, 8)
...
>>> checkNumOffspring(numOffspring=func)
[5, 7, 5, 5, 6, 2]
>>> # Case 3: A geometric distribution
>>> checkNumOffspring(numOffspring=(sim.GEOMETRIC_DISTRIBUTION, 0.3))
[3, 1, 1, 3, 4, 1, 1, 1, 2, 1, 1, 4, 6, 1]
>>> # Case 4: A Poisson distribution
>>> checkNumOffspring(numOffspring=(sim.POISSON_DISTRIBUTION, 1.6))
[2, 2, 1, 5, 3, 3, 1, 1, 2, 3, 3, 2, 2]
>>> # Case 5: A Binomial distribution
>>> checkNumOffspring(numOffspring=(sim.BINOMIAL_DISTRIBUTION, 0.1, 10))
[1, 4, 1, 1, 2, 1, 1, 3, 1, 1, 1, 3, 2, 2, 1, 1, 1, 2, 1]
>>> # Case 6: A uniform distribution
>>> checkNumOffspring(numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 6))
[4, 4, 2, 6, 6, 2, 2, 2, 2]
>>> # Case 7: With selection on offspring
>>> checkNumOffspring(numOffspring=8,
...     ops=[sim.MapSelector(loci=0, fitness={(0,0):1, (0,1):0.8, (1,1):0.5})])

```

(continues on next page)

(continued from previous page)

```
[8, 5, 7, 6, 4]
```

```
now exiting runScriptInteractively...
```

Download `numOff.py`

However, **the actual number of offspring can be less than specified because offspring can be discarded during mating**. More specifically, if any during- mating generator, such as a during-mating selector, returns `False` during the production of offspring, the offspring will be discarded so the total number of offspring will be reduced. This is the case in the seventh case of Example `numOff` where offspring with certain genotypes have lower probabilities to survive. If you would like to control size of families in the presence of natural selection, you could set a larger `numOffspring` use a `OffspringTagger` to mark the index of offspring, and discard offspring conditionally using operator `DiscardIf`. Please refer to example `OffspringTagger` for details.

6.1.4 Dynamic population size determined by number of offspring *

What we have described so far requires you to determine the size of offspring population in advance. Each mating event produces a number of offspring that is determined by parameter `NumOffspring`. The mating process stops when the offspring population is filled. This works for most scenarios but there are cases where the offspring population size is determined dynamically from a fixed number of mating events with random number of offspring. For example, you might design a mating scheme where all males in a population mate only once and produce random number of offspring.

These kind of mating schemes can be simulated using a demographic model that calculates offspring population size from pre-simulated number of offspring for each family. More specifically, we

- Define a demographic function (model) that will be called before mating happens.
- This function determines and save the number of offspring for each mating event, and return the total number of offspring as offspring population size.
- Pass a function or generator to parameter `numOffspring` to pass pre-determined number of offspring. This function will be called each time when number of offspring is needed.

The number of offspring could be saved and retrieved as global variable but a more clever method is to store the numbers of offspring in a demographic model (class). Example `dynamicNumOff` demonstrates this method by implementing a demographic model that simulate, save, and return the number of offspring. Note that although we determine the number of mating events from number of males in the parental population, a random mating scheme will choose parents with replacement so it is likely that some parents will be chosen multiple times while some others are not chosen at all. Please refer to section “Non-random and customized mating schemes” to learn how to define a mating scheme that picks parents without replacement.

Example: *Dynamic population size determined by number of offspring*

```
>>> import simuPOP as sim
>>>
>>> import random
>>>
>>> class RandomNumOff:
...     # a demographic model
...     def __init__(self):
...         self.numOff = []
...
...     def getNumOff(self):
...         # return the pre-simulated number of offspring as a generator function
...         for item in self.numOff:
```

(continues on next page)

(continued from previous page)

```

...         yield item
...
...     def __call__(self, pop):
...         # define __call__ so that a RandomNumOff object is callable.
...         #
...         # Each male produce from 1 to 3 offspring. For large population, get the
...         # number of males instead of checking the sex of each individual
...         self.numOff = [random.randint(1, 3) for ind in pop.individuals() if ind.
↪sex() == sim.MALE]
...         # return the total population size
...         print('{} mating events with number of offspring {}'.format(len(self.
↪numOff), self.numOff))
...         return sum(self.numOff)
...
>>>
>>> pop = sim.Population(10)
>>>
>>> # create a demogranic model
>>> numOffModel = RandomNumOff()
>>>
>>> pop.evolve(
...     preOps=sim.InitSex(),
...     matingScheme=sim.RandomMating(
...         # the model will be called before mating to deteremine
...         # family and population size
...         subPopSize=numOffModel,
...         # the getNumOff function (generator) returns number of offspring
...         # for each mating event
...         numOffspring=numOffModel.getNumOff
...     ),
...     gen=3
... )
5 mating events with number of offspring [3, 2, 2, 3, 3]
6 mating events with number of offspring [3, 2, 3, 1, 2, 3]
6 mating events with number of offspring [2, 1, 1, 2, 3, 2]
3
>>>
now exiting runScriptInteractively...

```

[Download dynamicNumOff.py](#)

6.1.5 Determine sex of offspring

Because sex can influence how genotypes are transmitted (e.g. sex chromosomes, haplodiploid population), simuPOP determines offspring sex before it passes an offspring to a *genotype transmitter* (during-mating operator) to transmit genotype from parents to offspring. The default `sexMode` in almost all mating schemes is `RandomSex`, in which case simuPOP assign Male or Female to offspring with equal probability.

Other sex determination methods are also available:

- `sexMode=RANDOM_SEX`: Sex is determined randomly, with equal probability for MALE and FEMALE. This is the default mode for sexual mating schemes such as random mating.
- `sexMode=NO_SEX`: Sex is not simulated so everyone is MALE. This is the default mode for asexual mating schemes.

- NumOfMales and NumOfFemales are useful in theoretical studies where the sex ratio of a population needs to be controlled strictly, or in special mating schemes, usually for animal populations, where only a certain number of male or female Individuals are allowed in a family. It worth noting that a genotype transmitter can override specified offspring sex. This is the case for *CloneGenoTransmitter* where an offspring inherits both genotype and sex from his/her parent.

Example: *Determine the sex of offspring*

(continues on next page)

(continued from previous page)

[illegible]

Download `sexMode.py`

6.1.6 Monogamous mating

Monogamous mating (monogamy) in simuPOP refers to mating schemes in which each parent mates only once. In an asexual setting, this implies parents are chosen without replacement. In sexual mating schemes, this means that parents are chosen without replacement, they have only one spouse during their life time so that all siblings have the same parents (no half-sibling).

simuPOP provides a diploid sexual monogamous mating scheme *MonogamousMating*. However, without careful planning, this mating scheme can easily stop working due to the lack of parents. For example, if a population has 40 males and 55 females, only 40 successful mating events can happen and result in 40 offspring in the offspring generation. *MonogamousMating* will exit if the offspring generation is larger than 40.

Example *monogamous* demonstrates one scenario of using a monogamous mating scheme where sex of parents and offspring are strictly specified so that parents will not be exhausted. The sex initializer *InitSex* assigns exactly 10 males and 10 females to the initial population. Because of the use of `numOffspring=2`, `sexMode=(NUM_OF_MALES, 1)`, each mating event will produce exactly one male and one female. Unlike a random mating scheme that only about 80% of parents are involved in the production of an offspring population with the same size, this mating scheme makes use of all parents.

Example: *Sexual monogamous mating*

```
>>> import simuPOP as sim
>>> pop = sim.Population(20, infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(sex=(sim.MALE, sim.FEMALE)),
...     matingScheme=sim.MonogamousMating(
...         numOffspring=2,
...         sexMode=(sim.NUM_OF_MALES, 1),
```

(continues on next page)

(continued from previous page)

```

...     ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.ParentsTagger(),
...     ],
... ),
...     gen = 5
... )
5
>>> [ind.sex() for ind in pop.individuals()]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> [int(ind.father_idx) for ind in pop.individuals()]
[16, 16, 2, 2, 4, 4, 8, 8, 0, 0, 14, 14, 10, 10, 12, 12, 18, 18, 6, 6]
>>> [int(ind.mother_idx) for ind in pop.individuals()]
[13, 13, 17, 17, 1, 1, 15, 15, 19, 19, 9, 9, 3, 3, 5, 5, 7, 7, 11, 11]
>>> # count the number of distinct parents
>>> len(set(pop.indInfo('father_idx')))
10
>>> len(set(pop.indInfo('mother_idx')))
10

now exiting runScriptInteractively...

```

[Download monogamous.py](#)

6.1.7 Polygamous mating

In comparison to monogamous mating, parents in a polygamous mate with more than one spouse during their life-cycle. Both *polygamy* (one man has more than one wife) and *polyandry* (one woman has more than one husband) are supported.

Other than regular parameters such as numOffspring, mating scheme `PolygamousMating` accepts parameters `polySex` (default to `Male`) and `polyNum` (default to 1). During mating, an individual with `polySex` is selected and then mate with `polyNum` randomly selected spouse. Example *polygamous* demonstrates the use of this mating schemes. Note that this mating scheme support natural selection, but does not yet handle varying `polyNum` and selection of parents without replacement.

Example: *Sexual polygamous mating*

```

>>> import simuPOP as sim
>>> pop = sim.Population(100, infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.PolygamousMating(polySex=sim.MALE, polyNum=2,
...     ops=[sim.ParentsTagger(),
...         sim.MendelianGenoTransmitter()]),
...     ),
...     gen = 5
... )
5
>>> [int(ind.father_idx) for ind in pop.individuals()][:20]
[67, 67, 42, 42, 91, 91, 25, 25, 65, 65, 47, 47, 18, 18, 16, 16, 96, 96, 57, 57]
>>> [int(ind.mother_idx) for ind in pop.individuals()][:20]
[58, 58, 58, 0, 68, 32, 37, 89, 6, 85, 12, 58, 36, 12, 66, 44, 51, 85, 60, 29]

now exiting runScriptInteractively...

```

[Download polygamous.py](#)

6.1.8 Asexual random mating

Mating scheme *RandomSelection* implements an asexual random mating scheme. It randomly select parents from a parental population (with replacement) and copy them to an offspring generation. Both genotypes and sex of the parents are copied because genotype and sex are sometimes related. This mating scheme can be used to simulate the evolution of haploid sequences in a standard haploid Wright-Fisher model.

Example *RandomSelection* applies a *RandomSelection* mating scheme to a haploid population with 100 sequences. A parentTagger is used to track the parent of each individual. Although sex information is not used in this mating scheme, Individual sexes are initialized and passed to offspring.

Example: *Asexual random mating*

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, ploidy=1, loci=[5, 5], ancGen=1,
...     infoFields='parent_idx')
>>> pop.evolve(
...     initOps=sim.InitGenotype(freq=[0.3, 0.7]),
...     matingScheme=sim.RandomSelection(ops=[
...         sim.ParentsTagger(infoFields='parent_idx'),
...         sim.CloneGenoTransmitter(),
...     ]),
...     gen = 5
... )
5
>>> ind = pop.individual(0)
>>> par = pop.ancestor(ind.parent_idx, 1)
>>> print(ind.sex(), ind.genotype())
1 [1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
>>> print(par.sex(), par.genotype())
1 [1, 1, 0, 0, 1, 1, 1, 1, 0, 1]

now exiting runScriptInteractively...
```

[Download RandomSelection.py](#)

6.1.9 Mating in haplodiploid populations

Male individuals in a haplodiploid population are derived from unfertilized eggs and thus have only one set of chromosomes. Mating in such a population is handled by a special mating scheme called *haplodiploidMating*. This mating scheme chooses a pair of parents randomly and produces some offspring. It transmit maternal chromosomes and paternal chromosomes (the only copy) to female offspring, and only maternal chromosomes to male offspring. Example *HaplodiploidMating* demonstrates how to use this mating scheme. It uses three initializers because sex has to be initialized before two other initializers can initialize genotype by sex.

Example: *Random mating in haplodiploid populations*

```
>>> import simuPOP as sim
>>> pop = sim.Population(10, ploidy=sim.HAPLODIPLOID, loci=[5, 5],
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
```

(continues on next page)

(continued from previous page)

```

...     sim.InitGenotype(genotype=[0]*10, subPops=[(0, 'Male')]),
...     sim.InitGenotype(genotype=[1]*10+[2]*10, subPops=[(0, 'Female')])
... ],
... preOps=sim.Dumper(structure=False),
... matingScheme=sim.HaplodiploidMating(
...     ops=[sim.HaplodiploidGenoTransmitter(), sim.ParentsTagger()]),
... postOps=sim.Dumper(structure=False),
... gen = 1
... )
SubPopulation 0 (), 10 Individuals:
 0: FU 11111 11111 | 22222 22222 | 0 0
 1: FU 11111 11111 | 22222 22222 | 0 0
 2: MU 00000 00000 | _____ | 0 0
 3: MU 00000 00000 | _____ | 0 0
 4: MU 00000 00000 | _____ | 0 0
 5: MU 00000 00000 | _____ | 0 0
 6: MU 00000 00000 | _____ | 0 0
 7: FU 11111 11111 | 22222 22222 | 0 0
 8: FU 11111 11111 | 22222 22222 | 0 0
 9: FU 11111 11111 | 22222 22222 | 0 0

SubPopulation 0 (), 10 Individuals:
 0: MU 11111 11111 | _____ | 4 9
 1: MU 11111 22222 | _____ | 4 8
 2: MU 22222 11111 | _____ | 6 8
 3: MU 22222 11111 | _____ | 3 8
 4: MU 22222 22222 | _____ | 2 8
 5: MU 22222 22222 | _____ | 6 9
 6: FU 22222 22222 | 00000 00000 | 2 1
 7: FU 22222 22222 | 00000 00000 | 2 1
 8: FU 22222 22222 | 00000 00000 | 3 9
 9: FU 11111 11111 | 00000 00000 | 5 8

1

now exiting runScriptInteractively...

```

[Download HaplodiploidMating.py](#)

Note that this mating scheme does not support recombination and the standard Recombinator does not work with haplodiploid populations. Please refer to the next Chapter for how to define a customized genotype transmitter to handle such a situation.

6.1.10 Self-fertilization

Some plant populations evolve through self-fertilization. That is to say, a parent fertilizes with itself during the production of offspring (seeds). In a *SelfMating* mating scheme, parents are chosen randomly (one at a time), and are used twice to produce two homologous sets of offspring chromosomes. The standard Recombinator can be used with this mating scheme. Example *SelfMating* initializes each chromosome with different alleles to demonstrate how these alleles are transmitted in this population.

Example: *Selfing mating scheme*

```

>>> import simuPOP as sim
>>> pop = sim.Population(20, loci=8)

```

(continues on next page)

(continued from previous page)

```

>>> # every chromosomes are different. :-)
>>> for idx, ind in enumerate(pop.individuals()):
...     ind.setGenotype([idx*2], 0)
...     ind.setGenotype([idx*2+1], 1)
...
>>> pop.evolve(
...     matingScheme=sim.SelfMating(ops=sim.Recombinator(rates=0.01)),
...     gen = 1
... )
1
>>> sim.dump(pop, width=3, structure=False, max=10)
SubPopulation 0 (), 20 Individuals:
  0: FU  36 36 36 36 36 36 36 36 | 36 36 36 36 36 36 36 36
  1: FU   6  6  6  6  6  6  6  6 |  7  7  7  7  7  7  7  7
  2: MU  33 33 33 33 33 33 33 33 | 33 33 33 33 33 33 33 33
  3: MU  22 22 22 22 22 23 23 23 | 22 22 22 22 22 22 22 22
  4: FU  27 27 27 27 27 27 27 27 | 27 27 27 27 27 27 27 27
  5: MU  15 15 15 15 15 15 15 15 | 15 15 15 15 15 15 15 15
  6: MU  35 35 35 35 34 34 34 34 | 34 34 34 34 34 34 34 34
  7: FU  11 11 11 11 11 11 11 11 | 10 10 10 10 10 10 10 10
  8: MU  11 11 11 11 11 11 11 11 | 11 11 11 11 11 11 11 11
  9: FU  24 24 24 24 24 24 24 24 | 25 25 25 25 25 25 25 25

now exiting runScriptInteractively...

```

[Download SelfMating.py](#)

6.1.11 Heterogeneous mating schemes *

Different groups of individuals in a population may have different mating patterns. For example, individuals with different properties can have varying fecundity, represented by different numbers of offspring generated per mating event. This can be extended to aged populations in which only adults (may be defined by age > 20 and age < 40) can produce offspring, where other individuals will either be copied to the offspring generation or die.

A heterogeneous mating scheme (*HeteroMating*) accepts a list of mating schemes that are applied to different subpopulation or virtual subpopulations. If multiple mating schemes are applied to the same subpopulation, each of them only populate part of the offspring subpopulation. This is illustrated in Figure [fig_heterogenous_mating](#).

Figure: *Illustration of a heterogeneous mating scheme*

A heterogeneous mating scheme that applies homogeneous mating schemes MS0, MS0.0, MS0.1, MS1, MS2.0 and MS2.1 to subpopulation 0, the first and second virtual subpopulation in subpopulation 0, subpopulation 1, the first and second virtual subpopulation in subpopulation 2, respectively. Note that VSP 0 and 1 in subpopulation 0 overlap, and do not add up to subpopulation 0.

For example, Example [heteroMatingSP](#) applies two random mating schemes to two subpopulations. The first mating scheme produces two offspring per mating event, and the second mating scheme produces four.

Example: *Applying different mating schemes to different subpopulations*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000, 1000], loci=2,
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=sim.InitSex(),

```

(continues on next page)



Users/bpeng1/simuPOP/simuPOP/doc/figures/MatingScheme.png

(continued from previous page)

```

...     matingScheme=sim.HeteroMating([
...         sim.RandomMating(numOffspring=2, subPops=0,
...             ops=[sim.MendelianGenoTransmitter(), sim.ParentsTagger()])
...     ],
...     sim.RandomMating(numOffspring=4, subPops=1,
...         ops=[sim.MendelianGenoTransmitter(), sim.ParentsTagger()])
... )
... ],
... gen=10
... )
10
>>> [int(ind.father_idx) for ind in pop.individuals(0)][:10]
[134, 134, 451, 451, 780, 780, 443, 443, 457, 457]
>>> [int(ind.father_idx) for ind in pop.individuals(1)][:10]
[1978, 1978, 1978, 1978, 1582, 1582, 1582, 1582, 1322, 1322]

now exiting runScriptInteractively...

```

Download HeteroMatingSP.py

The real power of heterogeneous mating schemes lies on their ability to apply different mating schemes to different virtual subpopulations. For example, due to different micro-environmental factors, plants in the same population may exercise both self and cross-fertilization. Because of the randomness of such environmental factors, it is difficult to divide a population into self and cross-mating subpopulations. Applying different mating schemes to groups of individuals in the same subpopulation is more appropriate.

Example *heteroMatingVSP* applies two mating schemes to two VSPs defined by proportions of individuals. In this mating scheme, 20% of individuals go through self-mating and 80% of individuals go through random mating. This can be seen from the parental indexes of individuals in the offspring generation: individuals whose `mother_idx` are -1 are genetically only derived from their fathers.

It might be surprising that offspring resulted from two mating schemes mix with each other so the same VSPs in the next generation include both selfed and cross-fertilized offspring. If this not desired, you can set parameter `shuffleOffspring=False` in *HeteroMating*(). Because the number of offspring that are produced by each mating scheme is proportional to the size of parental (virtual) subpopulation, the first 20% of individuals that are produced by self-fertilization will continue to self-fertilize.

Example: *Applying different mating schemes to different virtual subpopulations*

```

>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000], loci=2,
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.setVirtualSplitter(sim.ProportionSplitter([0.2, 0.8]))
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.HeteroMating(matingSchemes=[
...         sim.SelfMating(subPops=[(0, 0)],
...             ops=[sim.SelfingGenoTransmitter(), sim.ParentsTagger()])
...     ],
...     sim.RandomMating(subPops=[(0, 1)],
...         ops=[sim.SelfingGenoTransmitter(), sim.ParentsTagger()])
... )
... ],
... gen = 10
... )
10
>>> [int(ind.father_idx) for ind in pop.individuals(0)][:15]

```

(continues on next page)

(continued from previous page)

```
[789, 666, 145, 125, 681, 183, 727, 308, 392, 11, 183, 223, 208, 29, 309]
>>> [int(ind.mother_idx) for ind in pop.individuals(0)][:15]
[370, 272, -1, 520, 121, 91, 220, 519, 101, 271, -1, 263, 663, -1, 286]

now exiting runScriptInteractively...
```

Download HeteroMatingVSP.py

Because there is no restriction on the choice of VSPs, mating schemes can be applied to overlapped (virtual) subpopulations. For example,

```
HeteroMating(
    matingSchemes = [
        SelfMating(subPops=[(0, 0)]),
        RandomMating(subPops=0)
    ]
)
```

will apply *SelfMating* to the first 20% individuals, and *RandomMating* will be applied to all individuals. Similarly,

```
HeteroMating(
    matingSchemes = [
        SelfMating(subPops=0),
        RandomMating(subPops=0)
    ]
)
```

will allow all individuals to be involved in both *SelfMating* and *RandomMating*.

This raises the question of how many offspring each mating scheme will produce. By default, the number of offspring produced will be proportional to the size of parental (virtual) subpopulations. In the last example, because both mating schemes are applied to the same subpopulation, half of all offspring will be produced by selfing and the other half will be produced by random mating.

This behavior can be changed by a weighting scheme controlled by parameter *weight* of each homogeneous mating scheme. Briefly speaking, a positive weight will be compared against other mating schemes. a negative weight is considered proportional to the existing (virtual) subpopulation size. Negative weights are considered before positive or zero weights.

This weighting scheme is best explained by an example. Assuming that there are three mating schemes working on the same parental subpopulation

- Mating scheme A works on the whole subpopulation of size 1000
- Mating scheme B works on a virtual subpopulation of size 500
- Mating scheme C works on another virtual subpopulation of size 800

Assuming the corresponding offspring subpopulation has individuals,

- If all weights are 0, the offspring subpopulation is divided in proportion to parental (virtual) subpopulation sizes. In this example, the mating schemes will produce , , individuals respectively.
- If all weights are negative, they are multiplied to their parental (virtual) subpopulation sizes. For example, weight (-1, -2, -0.5) will lead to sizes (1000, 1000, 400) in the offspring subpopulation. If in this case, an error will be raised.
- If all weights are positive, the number of offspring produced from each mating scheme is proportional to these weights. For example, weights (1, 2, 3) will lead to , , individuals respectively. In this case, 0 weights will produce no offspring.

- If there are mixed positive and negative weights, the negative weights are processed first, and the rest of the individuals are divided using non-negative weights. For example, three mating schemes with weights (-0.5, 2, 3) will produce 500, , individuals respectively.

The last case is demonstrated in Example [HeteroMatingWeight](#) where three random mating schemes are applied to subpopulation 0, virtual subpopulation (0, 0) and virtual subpopulation (0, 1), with weights -0.5, 2, and 3 respectively. This example uses an advanced features that will be described in the next section. Namely, three during-mating Python operators are passed to each mating scheme to mark their offspring with different numbers.

Example: *A weighting scheme used by heterogeneous mating schemes.*

```
>>> import simuPOP as sim
>>> pop = sim.Population(size=[1000], loci=2,
...     infoFields='mark')
>>> pop.setVirtualSplitter(sim.RangeSplitter([[0, 500], [200, 1000]]))
>>>
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     matingScheme=sim.HeteroMating([
...         sim.RandomMating(subPops=0, weight=-0.5,
...             ops=[sim.InfoExec('mark=0'), sim.MendelianGenoTransmitter()]),
...         sim.RandomMating(subPops=[0, 0], weight=2,
...             ops=[sim.InfoExec('mark=1'), sim.MendelianGenoTransmitter()]),
...         sim.RandomMating(subPops=[0, 1], weight=3,
...             ops=[sim.InfoExec('mark=2'), sim.MendelianGenoTransmitter()])
...     ]),
...     gen = 10
... )
10
>>> marks = list(pop.indInfo('mark'))
>>> marks.count(0.)
500
>>> marks.count(1.)
200
>>> marks.count(2.)
300

now exiting runScriptInteractively...
```

Download [HeteroMatingWeight.py](#)

As a special case that can be quite annoying during the simulation of small populations, a (virtual) subpopulation can have no male and/or female. If the parental (virtual) subpopulation is empty, it will produce no offspring regardless of its weight. However, if the parental (virtual) subpopulation is not empty, it will be expected to produce some offspring, which is not possible if a sexual mating scheme is used. In this case, you can use a parameter `weightBy` to specify how parental (virtual) population sizes are calculated. This parameter accepts values `ANY_SEX` (default), `MALE_ONLY`, `FEMALE_ONLY`, `PAIR_ONLY`, and use all individuals, number of male individuals, number of female individuals, and number of male/female pairs (basically the less of numbers of males and females) as the size of parental (virtual) subpopulation, respectively. When `weightBy=PAIR_ONLY` is used, parental (virtual) subpopulations with only males or females will appear to be empty and produce no offspring. Note that in this mode (also `MALE_ONLY`, `FEMALE_ONLY`), the perceived parental population sizes are no longer the actual parental population sizes so you might need to adjust parameter `weight` (e.g. `weight=-2`) to produce correct number of offspring.

6.1.12 Conditional mating schemes

A [ConditionalMating](#) mating scheme allows you to apply different mating schemes to populations with different properties. The condition can be a constant (True or False), an expression that will be evaluated in the local namespace

of the parental population, or a function that can take parental population as its input parameter (with parameter name `pop`).

Using variable `rep` and `gen` in the local namespace of the parental population, we can use this mating scheme to apply different mating schemes to different replicates and/or at different generations. For example, *[matingSchemeByRepAndGen](#)* simulates the evolution of three replicates. The first replicate uses regular mating scheme, the third replicate uses a mating scheme that produces 70% of males, and the second replicate do this only for the first 5 generations. Because there are three cases, a nested *[ConditionalMating](#)* is used.

Example: Apply different mating schemes for different replicates at different generations

```
>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(1000, loci=[10]), rep=3)
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.ConditionalMating('rep == 0',
...         # the first replicate use standard random mating
...         sim.RandomMating(),
...         sim.ConditionalMating('rep == 1 and gen >= 5',
...             # the second replicate produces more males for the first 5 generations
...             sim.RandomMating(),
...             # the last replicate produces more males all the time
...             sim.RandomMating(sexMode=(sim.PROB_OF_MALES, 0.7))
...         )
...     ),
...     postOps=[
...         sim.Stat(numOfMales=True),
...         sim.PyEval("'gen=%d' % gen", reps=0),
...         sim.PyEval(r'\t%d' % numOfMales),
...         sim.PyOutput('\n', reps=-1)
...     ],
...     gen=10
... )
gen=0      477      686      718
gen=1      477      689      698
gen=2      519      692      713
gen=3      479      709      704
gen=4      539      710      688
gen=5      496      482      698
gen=6      489      488      701
gen=7      495      508      715
gen=8      497      488      688
gen=9      528      498      698
(10, 10, 10)

now exiting runScriptInteractively...
```

Download *[matingSchemeByRepAndGen.py](#)*

A function can be passed as the condition of a *[ConditionalMating](#)* mating scheme. This allows you to apply operators such as *[Stat](#)* to examine the condition of populations more closely and determine which mating scheme to use.

6.2 Simulator

A simuPOP simulator evolves one or more copies of a population forward in time, subject to various operators. Although a population could evolve by itself using function `Population.evolve`, a simulator with one replicate is actually used.

6.2.1 Add, access and remove populations from a simulator

A simulator could be created by one or more replicates of a list of populations. For example, you could create a simulator from five replicates of a population using

```
Simulator(pop, rep=5)
```

or from a list of populations using

```
Simulator([pop, pop1, pop2])
```

. `pop`, `pop1` and `pop2` do not have to have the same genotypic structure. In order to avoid duplication of potentially large populations, a population is by default *stolen* after it is used to create a simulator. If you would like to keep the populations, you could set parameter `stealPops` to `False` so that the populations will be copied to the simulator. Populations in a simulator can be added or removed using functions `Simulator.add()` and `Simulator.extract(idx)`.

When a simulator is created, you can access populations in this simulator using function `Simulator.population(idx)` or iterate through all populations using function `Simulator.populations()`. These functions return references to the populations so that you can access populations. Modifying these references will change the corresponding populations within the simulator. The references will become invalid once the simulator object is destroyed.

Example `Simulator` demonstrates different ways to create a simulator and how to access populations within it.

Example: *Create a simulator and access populations*

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=10)
>>> # five copies of the same population
>>> simu = sim.Simulator(pop, rep=5)
>>> simu.numRep()
5
>>> # evolve for ten generations and save the populations
>>> simu.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.3, 0.7])
...     ],
...     matingScheme=sim.RandomMating(),
...     finalOps=sim.SavePopulation('! "pop%d.pop"%rep'),
...     gen=10
... )
(10, 10, 10, 10, 10)
>>> # load the population and create another Simulator
>>> simu = sim.Simulator([sim.loadPopulation('pop%d.pop' % x) for x in range(5)])
>>> # continue to evolve
>>> simu.evolve(
...     matingScheme=sim.RandomMating(),
...     gen=10
```

(continues on next page)

(continued from previous page)

```

... )
(10, 10, 10, 10, 10)
>>> # print out allele frequency
>>> for pop in simu.populations():
...     sim.stat(pop, alleleFreq=0)
...     print('%.2f' % pop.dvars().alleleFreq[0][0])
...
0.36
0.30
0.28
0.01
0.11
>>> # get a population
>>> pop = simu.extract(0)
>>> simu.numRep()
4
now exiting runScriptInteractively...

```

[Download Simulator.py](#)

6.2.2 Number of generations to evolve

A simulator usually evolves a specific number of generations according to parameter `gen` of the `evolve` function. A generation number is used to track the number of generations a simulator has evolved. Because a new population has generation number 0, a population would be at the beginning of generation after it evolves generations. The generation number would increase if the simulator continues to evolve. During evolving, variables `rep` (replicate number) and `gen` (current generation number) are set to each population's local namespace.

It is not always possible to know in advance the number of generations to evolve. For example, you may want to evolve a population until a specific allele gets fixed or lost in the population. In this case, you can let the simulator run indefinitely (do not set the `gen` parameter) and depend on a **terminator** to terminate the evolution of a population. The easiest method to do this is to use population variables to track the status of a population, and use a *TerminateIf* operator to terminate the evolution according to the value of an expression. Example *simuGen* demonstrates the use of such a terminator, which terminates the evolution of a population if allele 0 at locus 5 is fixed or lost. It also shows the application of an interesting operator *IfElse*, which applies an operator, in this case *PyEval*, only when an expression returns `True`. Note that this example calls the `simulator.evolve` function twice. The first call does not specify a mating scheme so a default empty mating scheme (*MatingScheme*) that does not transmit genotype is used. Populations start from the beginning of the fifth generation when the second `simulator.evolve` function is called.

The generation number is stored in each Population using population variable `gen`. You can access these numbers from a simulator using function *Simulator.dvars*(idx) or from a population using function *Population.dvars*(). If needed, **you can reset generation numbers by changing these variables.**

Example: *Generation number of a simulator*

```

>>> import simuPOP as sim
>>> simu = sim.Simulator(sim.Population(50, loci=[10], ploidy=1),
...     rep=3)
>>> simu.evolve(gen = 5)
(5, 5, 5)
>>> simu.dvars(0).gen
5
>>> simu.evolve(

```

(continues on next page)

(continued from previous page)

```

...     initOps=[sim.InitGenotype(freq=[0.5, 0.5])],
...     matingScheme=sim.RandomSelection(),
...     postOps=[
...         sim.Stat(alleleFreq=5),
...         sim.IfElse('alleleNum[5][0] == 0',
...             sim.PyEval(r"'Allele 0 is lost in rep %d at gen %d\n' % (rep, gen)")),
...         sim.IfElse('alleleNum[5][0] == 50',
...             sim.PyEval(r"'Allele 0 is fixed in rep %d at gen %d\n' % (rep, gen)
...             ↪")),
...         sim.TerminateIf('len(alleleNum[5]) == 1'),
...     ],
... )
Allele 0 is fixed in rep 2 at gen 29
Allele 0 is fixed in rep 1 at gen 74
Allele 0 is lost in rep 0 at gen 120
(116, 70, 25)
>>> [simu.dvars(x).gen for x in range(3)]
[121, 75, 30]

now exiting runScriptInteractively...

```

[Download simuGen.py](#)

6.2.3 Evolve populations in a simulator

There are a number of rules about when and how operators are applied during the evolution of a population. In summary, in the order at which operators are processed and applied,

- Operators specified in parameter `initOps` of function `Simulator.evolve` will be applied to the initial population before evolution, subject to replicate applicability restraint specified by parameter `reps`.
- Operators specified in parameter `preOps` of function `Simulator.evolve` will be applied to the parental population at each generation, subject to replicate and generation applicability restraint specified by parameters `begin`, `end`, `step`, `at`, and `reps`.
- During-mating operators specified in the `ops` parameter of a mating scheme will be called during mating to transmit genotype (and possibly information fields etc) from parental to offspring, subject to replicate and generation applicability restraint specified by parameters `begin`, `end`, `step`, `at`, and `reps`.
- Operators specified in parameter `postOps` of function `Simulator.evolve` will be applied to the offspring population at each generation, subject to replicate and generation applicability restraint specified by parameters `begin`, `end`, `step`, `at`, and `reps`.
- Operators specified in parameter `finalOps` of function `Simulator.evolve` will be applied to the final population after evolution, subject to replicate applicability restraint specified by parameter `reps`.

Figure [fig_operator_orders](#) illustrated how operators are applied to an evolutionary process. It worth noting that a default during-mating operator is defined for each mating scheme. User-specified operators will **replace** the default operator so you need to explicitly specify the default operator if you intent to add another one.

Figure: *Orders at which operators are applied during an evolutionary process*

If you suspect that your simulation is not running as expected, you can have a close look at your evolutionary process by setting the `dryrun` parameter of an `evolve` function to `True`, or by calling function `describeEvolProcess()`. This function takes the same set of parameters as `Simulator.evolve()` and returns a description of the evolution process, which might help you identify misuse of operators.

Example: *describe an evolutionary process*



Users/bpeng1/simuPOP/simuPOP/doc/figures/operators.png

```

>>> import simuPOP as sim
>>>
>>> def outputstat(pop):
...     'Calculate and output statistics, ignored'
...     return True
...
>>> # describe this evolutionary process
>>> print(sim.describeEvolProcess(
...     initOps=[
...         sim.InitSex(),
...         sim.InitInfo(lambda: random.randint(0, 75), infoFields='age'),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.IdTagger(),
...         sim.PyOutput('Prevalence of disease in each age group:\n'),
...     ],
...     preOps=sim.InfoExec('age += 1'),
...     matingScheme=sim.HeteroMating([
...         sim.CloneMating(subPops=[(0,0), (0,1), (0,2)], weight=-1),
...         sim.RandomMating(ops=[
...             sim.IdTagger(),
...             sim.Recombinator(intensity=1e-4)
...         ], subPops=[(0,1)]),
...     ]),
...     postOps=[
...         sim.MaPenetrance(loci=0, penetrance=[0.01, 0.1, 0.3]),
...         sim.PyOperator(func=outputstat)
...     ],
...     gen = 100,
...     numRep = 3
... ))
Replicate 0 1 2:
Apply pre-evolution operators to the initial population (initOps).
* <simuPOP.InitSex> initialize sex randomly
* <simuPOP.InitInfo> initialize information field age using a Python
  function <lambda>
* <simuPOP.InitGenotype> initialize individual genotype according to
  allele frequencies.
* <simuPOP.IdTagger> assign an unique ID to individuals
* <simuPOP.PyOutput> write 'Prevalence of disease in each age group:... '
  to output

Evolve a population for 100 generations
* Apply pre-mating operators to the parental generation (preOps)
  # <simuPOP.InfoExec> execute statement age += 1 using information fields
  as variables.

* Populate an offspring population from the parental population using mating
  scheme <simuPOP.HeteroMating> a heterogeneous mating scheme with 2
  homogeneous mating schemes:
  # <simuPOP.HomoMating> a homogeneous mating scheme that uses
    - <simuPOP.SequentialParentChooser> chooses a parent sequentially
    - <simuPOP.OffspringGenerator> produces offspring using operators
      . <simuPOP.CloneGenoTransmitter> clone genotype, sex and
        information fields of parent to offspring
    in subpopulations (0, 0), (0, 1), (0, 2).
  # <simuPOP.HomoMating> a homogeneous mating scheme that uses
    - <simuPOP.RandomParentsChooser> chooses two parents randomly

```

(continues on next page)

(continued from previous page)

```

- <simuPOP.OffspringGenerator> produces offspring using operators
  . <simuPOP.IdTagger> assign an unique ID to individuals
  . <simuPOP.Recombinator> genetic recombination.
in subpopulations (0, 1).

* Apply post-mating operators to the offspring population (postOps).
  # <simuPOP.MaPenetrance> multiple-alleles penetrance
  # <simuPOP.PyOperator> calling a Python function outputstat

No operator is applied to the final population (finalOps).

now exiting runScriptInteractively...
```

Download [describe.py](#)

6.3 Non-random and customized mating schemes *

6.3.1 The structure of a homogeneous mating scheme *

A *homogeneous mating scheme* (*HomoMating*) populates an offspring generation as follows:

1. Create an empty offspring population (generation) with appropriate size. Parental and offspring generation can differ in size but they must have the same number of subpopulations.
2. For each subpopulation, repeatedly choose a parent or a pair of parents from the parental generation. This is done by a simuPOP object called a **parent chooser**.
3. One or more offspring are produced from the chosen parent(s) and are placed in the offspring population. This is done by a simuPOP **offspring generator**. An offspring generator uses one or more during-mating operators to transmit parental genotype to offspring. These operators are called **genotype transmitters**.
4. After the offspring generation is populated, it will replace the parental generation and becomes the present generation of a population.

To define a homogeneous mating scheme, you will need to provide a *chooser* (a *parent chooser* that is responsible for choosing one or two parents from the parental generation) and a *generator* (an *offspring generator* that is responsible for generating a number of offspring from the chosen parents). For example, a *selfingMating* mating scheme uses a *RandomParentChooser* to choose a parent randomly from a population, possibly according to individual fitness, it uses a standard *OffspringGenerator* that uses a *selfingOffspringGenerator* to transmit genotype. The constructor of *HomoMating* also accepts parameters *subPopSize* (parameter to control offspring subpopulation sizes), *subPops* (applicable subpopulations or virtual subpopulations), and *weight* (weighting parameter when used in a heterogeneous mating scheme). When this mating scheme is applied to the whole population, *subPopSize* is used to determine the subpopulation sizes of the offspring generation (see Section *subsec_offspring_size* for details), parameters *subPops* and *weight* are ignored. Otherwise, the number of offspring this mating scheme will produce is determined by the heterogeneous mating scheme.

Example *RandomMating* demonstrates how the most commonly used mating scheme, the diploid sexual *RandomMating* mating scheme is defined in *simuPOP.py*. This mating scheme uses a *RandomParentsChooser* with replacement, and a standard *OffspringGenerator* using a default *MendelianGenoTransmitter*.

Example: Define a random mating scheme

```
def RandomMating(numOffspring=1., sexMode=RANDOM_SEX,
                 ops=MendelianGenoTransmitter(), subPopSize=[],
                 subPops=ALL_AVAIL, weight=0, selectionField='fitness'):
    'A basic diploid sexual random mating scheme.'
    return HomoMating(
        chooser=RandomParentsChooser(True, selectionField),
        generator=OffspringGenerator(ops, numOffspring, sexMode),
        subPopSize=subPopSize,
        subPops=subPops,
        weight=weight)
```

Download RandomMating.py

Different parent choosers and offspring generators can be combined to define a large number of homogeneous mating schemes. Some of the parent choosers return one parent so they work with offspring generators that need one parent (e.g. selfing or clone offspring generator); some of the parent choosers return two parents so they work with offspring generators that need two parents (e.g. Mendelian offspring generator). For example, the standard *SelfMating* mating scheme uses a *RandomParentChooser* but you can easily use a *SequentialParentChooser* to choose parents sequentially and self-fertilize parents one by one. This is demonstrated in Example *sequentialSelfing*.

Example: Define a sequential selfing mating scheme

```
>>> import simuPOP as sim
>>> pop = sim.Population(100, loci=5*3, infoFields='parent_idx')
>>> pop.evolve(
...     initOps=[sim.InitGenotype(freq=[0.2]*5)],
...     preOps=sim.Dumper(structure=False, max=5),
...     matingScheme=sim.HomoMating(
...         sim.SequentialParentChooser(),
...         sim.OffspringGenerator(ops=[
...             sim.SelfingGenoTransmitter(),
...             sim.ParentsTagger(infoFields='parent_idx'),
...         ])
...     ),
...     postOps=sim.Dumper(structure=False, max=5),
...     gen = 1
... )
SubPopulation 0 (), 100 Individuals:
  0: MU 441000142224423 | 431303440010114 | 0
  1: MU 334442443034342 | 113203441333201 | 0
  2: MU 034344042424240 | 344304121430212 | 0
  3: MU 132322330420043 | 141300223114240 | 0
  4: MU 111123040033342 | 344344221133120 | 0

SubPopulation 0 (), 100 Individuals:
  0: MU 441000142224423 | 431303440010114 | 0
  1: FU 334442443034342 | 113203441333201 | 1
  2: MU 344304121430212 | 034344042424240 | 2
  3: FU 141300223114240 | 132322330420043 | 3
  4: FU 344344221133120 | 111123040033342 | 4

1

now exiting runScriptInteractively...
```

Download sequentialSelfing.py

6.3.2 Offspring generators *

An *OffspringGenerator* accepts a parameters *ops* (a list of during- mating operators), *numOffspring* (control number of offspring per mating event) and *sexMode* (control offspring sex). We have examined the last two parameters in detail in sections *subsec_number_of_offspring* and *subsec_offspring_sex*.

The most tricky parameter is the *ops* parameter. It accepts a list of during mating operators that are used to transmit genotypes from parent(s) to offspring and/or set individual information fields. The standard *OffspringGenerator* does not have any default operator so no genotype will be transmitted by default. All stock mating schemes use a default genotype transmitter. (e.g, a *MendelianGenoTransmitter* in Example *RandomMating* is passed to the offspring generator used in *RandomMating*). Note that you need to specify all needed operators if you use parameter *ops* to change the operators used in a mating scheme (see Example *HeteroMatingWeight*). That is to say, you can use *ops=Recombinator()* to replace a default *MendelianGenoTransmitter()*, but you have to use *ops=[IdTagger(), MendelianGenoTransmitter()]* if you would like to add a during-mating operator to the default one.

Another offspring generator is provided in simuPOP. This *ControlledOffspringGenerator* is used to control an evolutionary process so that the allele frequencies at certain loci follows some pre-simulated *frequency trajectories*. Please refer to Peng2007a for rationals behind such an offspring generator and its applications in the simulation of complex human diseases.

Example *controlledOffGenerator* demonstrates the use of such a controlled offspring generator. Instead of using a realistic frequency trajectory function, it forces allele frequency at locus 5 to increase linearly. In contrast, the allele frequency at locus 15 on the second chromosome oscillates as a result of genetic drift. Note that the random mating version of this mating scheme is defined in simuPOP as *ControlledRandomMating*.

Example: *A controlled random mating scheme*

```
>>> import simuPOP as sim
>>> def traj(gen):
...     return [0.5 + gen * 0.01]
...
>>> pop = sim.Population(1000, loci=[10]*2)
>>> # evolve the sim.Population while keeping allele frequency 0.5
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.HomoMating(sim.RandomParentChooser(),
...         sim.ControlledOffspringGenerator(loci=5,
...             alleles=[0], freqFunc=traj,
...             ops = sim.SelfingGenoTransmitter())),
...     postOps=[
...         sim.Stat(alleleFreq=[5, 15]),
...         sim.PyEval(r'("%.2f\t%.2f\n" % (alleleFreq[5][0], alleleFreq[15][0]))')
...     ],
...     gen = 5
... )
0.50 0.51
0.51 0.51
0.52 0.51
0.53 0.52
0.54 0.54
5
now exiting runScriptInteractively...
```

[Download controlledOffGenerator.py](#)

6.3.3 Genotype transmitters *

Although any during mating operators can be used in parameter `ops` of an offspring generator, those that transmit genotype from parents to offspring are customarily called **genotype transmitters**. `simuPOP` provides a number of genotype transmitters including `clone`, `Mendelian`, `selfing`, `haplodiploid`, `genotype transmitter`, and a `Recombinator`. They are usually used implicitly in a mating scheme, but they can also be used explicitly.

Although `simuPOP` provides a number of genotype transmitters, they may still be cases where customized genotype transmitter is needed. For example, a `Recombinator` can be used to recombine parental chromosomes but it is well known that male and female individuals differ in recombination rates. How can you apply two different `Recombinators` to male and female Individuals separately?

An immediate thought can be the use of virtual subpopulations. If you apply two random mating schemes to two virtual subpopulations defined by sex, `RandomParentsChooser` will not work because no opposite sex can be found in each virtual subpopulation. In this case, a customized genotype transmitter can be used.

A customized genotype transmitter is only a Python during-mating operator. Although it is possible to define a function and use a `PyOperator` directly (Example `PyOperator`), it is much better to derive an operator from `PyOperator`, as the case in Example `newOperator`.

Example `sexSpecificRec` defines a `sexSpecificRecombinator` that uses, internally, two different `Recombinators` to recombine male and female parents. The key statement is the `PyOperator.__init__` line which initializes a Python operator with given function `self.transmitGenotype`. Example `sexSpecificRec` outputs the population in two generations. You should notice that paternal chromosome are not recombined when they are transmitted to offspring.

Example: *A customized genotype transmitter for sex-specific recombination*

```
>>> from simuPOP import *
>>> class sexSpecificRecombinator(PyOperator):
...     def __init__(self, intensity=0, rates=0, loci=[], convMode=NO_CONVERSION,
...                 maleIntensity=0, maleRates=0, maleLoci=[], maleConvMode=NO_CONVERSION,
...                 *args, **kwargs):
...         # This operator is used to recombine maternal chromosomes
...         self.Recombinator = Recombinator(rates, intensity, loci, convMode)
...         # This operator is used to recombine paternal chromosomes
...         self.maleRecombinator = Recombinator(maleRates, maleIntensity,
...         maleLoci, maleConvMode)
...         #
...         PyOperator.__init__(self, func=self.transmitGenotype, *args, **kwargs)
...     #
...     def transmitGenotype(self, pop, off, dad, mom):
...         # Form the first homologous copy of offspring.
...         self.Recombinator.transmitGenotype(mom, off, 0)
...         # Form the second homologous copy of offspring.
...         self.maleRecombinator.transmitGenotype(dad, off, 1)
...         return True
...
>>> pop = Population(10, loci=[15]*2, infoFields=['father_idx', 'mother_idx'])
>>> pop.evolve(
...     initOps=[
...         InitSex(),
...         InitGenotype(freq=[0.4] + [0.2]*3)
...     ],
...     matingScheme=RandomMating(ops=[
...         sexSpecificRecombinator(rates=0.1, maleRates=0),
...         ParentsTagger()
...     ]),
```

(continues on next page)

(continued from previous page)

```

...     postOps=Dumper(structure=False),
...     gen = 2
... )
SubPopulation 0 (), 10 Individuals:
  0: FU 230000130212000 130110020112120 | 310300000030330 000113003202000 | 6 7
  1: FU 110100000002000 223313300111002 | 331311301000220 002330110020020 | 6 7
  2: MU 230301121003012 032010332330303 | 303303022100031 310232031321031 | 5 0
  3: MU 103001320130222 031300110100023 | 303303022100031 003000012020002 | 5 9
  4: FU 210230113000000 231111000121000 | 303303022100031 003000012020002 | 5 8
  5: MU 322030133101023 110323303020211 | 322111021000001 301200303300133 | 2 8
  6: MU 210230113000000 231111000121000 | 331303300011323 310232031321031 | 5 8
  7: FU 200331312001001 200011203020203 | 031032120003212 101032020302120 | 3 1
  8: FU 230000130212000 223313300111002 | 303303022100031 003000012020002 | 5 7
  9: FU 200331312001001 130301011230300 | 322111021000001 320103032303101 | 2 1

SubPopulation 0 (), 10 Individuals:
  0: MU 230000130212000 223313300111002 | 322030133101023 301200303300133 | 5 8
  1: MU 230000130212000 130110020112120 | 303303022100031 310232031321031 | 2 0
  2: FU 303303022100031 003000012020002 | 322111021000001 301200303300133 | 5 4
  3: FU 331311301000220 223313300111002 | 322111021000001 110323303020211 | 5 1
  4: MU 200331312001001 101032020302120 | 230301121003012 032010332330303 | 2 7
  5: FU 031032120003212 200011203020203 | 103001320130222 031300110100023 | 3 7
  6: FU 200331312001001 320103032303101 | 303303022100031 032010332330303 | 2 9
  7: FU 200331312001001 320103032303101 | 303303022100031 310232031321031 | 2 9
  8: FU 200331312001001 130301011230300 | 303303022100031 031300110100023 | 3 9
  9: MU 303303022100031 003000012020002 | 210230113000000 231111000121000 | 6 4

2

now exiting runScriptInteractively...

```

[Download sexSpecificRec.py](#)

6.3.4 A Python parent chooser *

Parent choosers are responsible for choosing one or two parents from a parental (virtual) subpopulation. simuPOP defines a few parent choosers that choose parent(s) sequentially, randomly (with or without replacement), or with additional conditions. Some of these parent choosers support natural selection. We have seen sequential and random parent choosers in Examples *sequentialSelfing* and *controlledOffGenerator*. Please refer to the simuPOP reference manual for details about these objects.

A parent choosing scheme can be quite complicated in reality. For example, salamanders along a river may mate with their neighbors and form several subspecies. This behavior cannot be readily simulated using any pre-define parent choosers so a hybrid parent chooser *PyParentsChooser*() should be used.

A *PyParentsChooser* accepts a user-defined Python generator function, instead of a normal python function, that returns a parent, or a pair of parents repeatedly. Briefly speaking, when a generator function is called, it returns a *generator* object that provides an iterator interface. Each time when this iterator iterates, this function resumes where it was stopped last time, executes and returns what the next *yield* statement returns. For example, example *generator* defines a function that calculate for . It does not calculate each repeatedly but returns , , ... sequentially.

Example: A sample generator function

```

>>> import simuPOP as sim
>>> def func():

```

(continues on next page)

(continued from previous page)

```

...     i = 1
...     all = 0
...     while i <= 5:
...         all += 1./i
...         i += 1
...         yield all
...
>>> for i in func():
...     print('%.3f' % i)
...
1.000
1.500
1.833
2.083
2.283

now exiting runScriptInteractively...

```

Download generator.py

A *PyParentsChooser* accepts a parent generator function, which takes a population and a subpopulation index as parameters. When this parent chooser is applied to a subpopulation, it will call this generator function and ask the generated generator object repeated for either a parent, or a pair of parents (*references to individual objects or indexes relative to a subpopulation*). Note that *PyParentsChooser* does not support virtual subpopulation but you can mimic the effect by returning only parents from certain virtual subpopulations.

Example *PyParentsChooser* implements a hybrid parent chooser that chooses parents with equal social status (rank). In this parent chooser, all males and females are categorized by their sex and social status. A parent is chosen randomly, and then his/her spouse is chosen from females/males with the same social status. The rank of their offspring can increase or decrease randomly. It becomes obvious now that whereas a python function can return random male/female pair, the generator interface is much more efficient because the identification of sex/status groups is done only once.

Example: A hybrid parent chooser that chooses parents by their social status

```

>>> import simuPOP as sim
>>> from random import randint
>>> def randomChooser(pop, subPop):
...     males = []
...     females = []
...     # identify males and females in each social rank
...     for rank in range(3):
...         males.append([x for x in pop.individuals(subPop) \
...             if x.sex() == sim.MALE and x.rank == rank])
...         females.append([x for x in pop.individuals(subPop) \
...             if x.sex() == sim.FEMALE and x.rank == rank])
...     #
...     while True:
...         # choose a rank randomly
...         rank = int(pop.individual(randint(0, pop.subPopSize(subPop) - 1), subPop).
... ↪rank)
...         yield males[rank][randint(0, len(males[rank]) - 1)], \
...             females[rank][randint(0, len(females[rank]) - 1)]
...
>>> def setRank(rank):
...     'The rank of offspring can increase or drop to zero randomly'
...     # only use rank of the father
...     return (rank[0] + randint(-1, 1)) % 3

```

(continues on next page)

(continued from previous page)

```

...
>>> pop = sim.Population(size=[1000, 2000], loci=1, infoFields='rank')
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitInfo(lambda : randint(0, 2), infoFields='rank')
...     ],
...     matingScheme=sim.HomoMating(
...         sim.PyParentsChooser(randomChooser),
...         sim.OffspringGenerator(ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.PyTagger(setRank),
...         ])
...     ),
...     gen = 5
... )
5

now exiting runScriptInteractively...

```

Download PyParentsChooser.py

Built-in parent choosers could be used in a *PyParentsChooser* to choose parents. The parent chooser needs to be initialized with the parental population and subpopulation index. Calling the `chooseParents` function repeatedly will return pairs of individuals from the population (None will be returned for one of the parents if the parent chooser only returns one parent). The use of built-in parent choosers can improve the performance of your *PyParentsChooser*, especially for complex selection patterns (e.g. with natural selection). For example, *BuiltInParentsChooser* implements a similar mating scheme as Example *PyParentsChooser* but uses a *RandomParentChooser* to choose males randomly.

Example: Use built-in parent choosers in a Python parent chooser

```

>>> import simuPOP as sim
>>> from random import randint
>>>
>>> def randomChooser(pop, subPop):
...     maleChooser = sim.RandomParentChooser(sexChoice=sim.MALE_ONLY)
...     maleChooser.initialize(pop, subPop)
...     females = []
...     # identify females in each social rank
...     for rank in range(3):
...         females.append([x for x in pop.individuals(subPop) \
...             if x.sex() == sim.FEMALE and x.rank == rank])
...     #
...     while True:
...         # choose a random male
...         m = maleChooser.chooseParents()[0]
...         rank = int(m.rank)
...         # find a female in the same rank
...         yield m, females[rank][randint(0, len(females[rank]) - 1)]
...
>>> def setRank(rank):
...     'The rank of offspring can increase or drop to zero randomly'
...     # only use rank of the father
...     return (rank[0] + randint(-1, 1)) % 3
...
>>> pop = sim.Population(size=[1000, 2000], loci=1, infoFields='rank')

```

(continues on next page)

(continued from previous page)

```

>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitInfo(lambda : randint(0, 2), infoFields='rank')
...     ],
...     matingScheme=sim.HomoMating(
...         sim.PyParentsChooser(randomChooser),
...         sim.OffspringGenerator(ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.PyTagger(setRank),
...         ])
...     ),
...     gen = 5
... )
5
now exiting runScriptInteractively...

```

[Download BuiltInParentsChooser.py](#)

6.3.5 Using C++ to implement a parent chooser **

A user defined parent chooser can be fairly complex and computationally intensive. For example, if a parent tends to find a spouse in his/her vicinity, geometric distances between all qualified individuals and a chosen parent need to be calculated for each mating event. If the optimization of the parent chooser can speed up the simulation significantly, it may be worthwhile to write the parent chooser in C++.

Although it is feasible, and sometimes easier to derive a class from class `ParentChooser` in `mating.h` (.cpp), modifying simuPOP source code is not recommended because you would have to modify a new version of simuPOP whenever you upgrade your simuPOP distribution. Implementing your parent choosing algorithm in another Python module is preferred.

The first step is to write your own parent chooser in C/C++. Basically, you will need to pass all necessary information to the C++ level and implement an algorithm to choose parents randomly. Although simple function based solutions are possible, a C++ level class such as the `myParentsChooser` class defined in Example [parentChoose-Header](#) is recommended. This class is initialized with indexes of male and female individuals and use a function `chooseParents` to return a pair of parents randomly. This parent chooser is very simple but more complicated parent selection scenarios can be implemented similarly.

Example: *Implement a parent chooser in C++*

```

#include <stdlib.h>
#include <vector>
#include <utility>
using std::pair;
using std::vector;
class myParentsChooser
{
public:
    // A constructor takes all locations of male and female.
    myParentsChooser(const std::vector<int> & m, const std::vector<int> & f)
        : male_idx(m), female_idx(f)
    {
        srand(time(0));
    }
}

```

(continues on next page)

(continued from previous page)

```

pair<unsigned long, unsigned long> chooseParents()
{
    unsigned long male = rand() % male_idx.size();
    unsigned long female = rand() % male_idx.size();
    return std::make_pair(male, female);
}
private:
    vector<int> male_idx;
    vector<int> female_idx;
};

```

Download [myParentsChooser.h](#)

The second step is to wrap your C++ functions and classes to a Python module. There are many tools available but SWIG (www.swig.org) is arguably the most convenient and powerful one. To use SWIG, you will need to prepare an interface file, which basically tells SWIG which functions and classes you would like to expose and how to pass parameters between Python and C++. Example [parentsChooserInterface](#) lists an interface file for the C++ class defined in Example [parentChooseHeader](#). Please refer to the SWIG reference manual for details.

Example: *An interface file for the myParentsChooser class*

```

%module myParentsChooser
%{
#include "myParentsChooser.h"
%}
// std_vector.i for std::vector
#include "std_vector.i"
%template() std::vector<int>;
// stl.i for std::pair
#include "stl.i"
%template() std::pair<unsigned long, unsigned long>;
#include "myParentsChooser.h"

```

Download [myParentsChooser.i](#)

The exact procedure to generate and compile a wrapper file varies from system to system, and from compiler to compiler. Fortunately, the standard Python module setup process supports SWIG. All you need to do is to write a Python `setup.py` file and let the `distutils` module of Python handle all the details for you. A typical `setup.py` file is demonstrated in Example [parentsChooserSetup](#).

Example: *Building and installing the myParentsChooser module*

```

from distutils.core import setup, Extension
import sys
# Under linux/gcc, lib stdc++ is needed for C++ based extension.
if sys.platform == 'linux2':
    libs = ['stdc++']
else:
    libs = []
setup(name = "myParentsChooser",
      description = "A sample parent chooser",
      py_modules = ['myParentsChooser'], # will be generated by SWIG
      ext_modules = [
          Extension('_myParentsChooser',
                    sources = ['myParentsChooser.i'],
                    swig_opts = ['-O', '-shadow', '-c++', '-keyword', ],

```

(continues on next page)

(continued from previous page)

```

        include_dirs = ["."],
    )
]
)

```

Download setup.py

You parent chooser can now be compiled and installed using the standard Python `setup.py` commands such as

```
python setup.py install
```

Please refer to the Python reference manual for other building and installation options. Note that Python 2.4 and earlier do not support option `swig_opts` well so you might have to pass these options using command

```
python setup.py build_ext --swig-opts=-O -templatereduce \
    -shadow -c++ -keyword -nodefaultctor install
```

Example [parentChooseHeader](#) demonstrates how to use such a C++ parents chooser in your simuPOP script. It uses the same Python parent chooser interface as in [PyParentsChooser](#), but leaves all the (potentially) computationally intensive parts to the C++ level `myParentsChooser` object.

Example: *Implement a parent chooser in C++*

```

import simuPOP as sim

# The class myParentsChooser is defined in module myParentsChooser
try:
    from myParentsChooser import myParentsChooser
except ImportError:
    # if failed to import the C++ version, use a Python version
    import random
    class myParentsChooser:
        def __init__(self, maleIndexes, femaleIndexes):
            self.maleIndexes = maleIndexes
            self.femaleIndexes = femaleIndexes
        def chooseParents(self):
            return self.maleIndexes[random.randint(0, len(self.maleIndexes)-1)], \
                self.femaleIndexes[random.randint(0, len(self.femaleIndexes)-1)]

def parentsChooser(pop, sp):
    'How to call a C++ level parents chooser.'
    # create an object with needed information (such as x, y) ...
    pc = myParentsChooser(
        [x for x in range(pop.popSize()) if pop.individual(x).sex() == sim.MALE],
        [x for x in range(pop.popSize()) if pop.individual(x).sex() == sim.FEMALE])
    while True:
        # return indexes of parents repeatedly
        yield pc.chooseParents()

pop = sim.Population(100, loci=1)
simu.evolve(
    initOps=[
        sim.InitSex(),
        sim.InitGenotype(freq=[0.5, 0.5])
    ],
    matingScheme=sim.HomoMating(sim.PyParentsChooser(parentsChooser),
        sim.OffspringGenerator(ops=sim.MendelianGenoTransmitter()))),

```

(continues on next page)

(continued from previous page)

```

gen = 100
)

```

Download [cppParentChooser.py](#)

6.4 Age structured populations with overlapping generations **

Age is an important factor in many applications because it is related to many genetic (most obviously mating) and environmental factors that influence the evolution of a population. The evolution of age structured populations will lead to overlapping generations because parents can co-exist with their offspring in such a population. Although simuPOP is based on a discrete generation model, it can be used to simulate age structured populations.

To evolve an age structured population, you will need to

- Define an information field `age` and use it to store age of all individuals. Age is usually assigned randomly at the beginning of a simulation.
- Define a virtual splitter that splits the parental population into several virtual subpopulation. The most important VSP consists of mating individuals (e.g. individuals with age between 20 and 40). Advanced features of virtual splitters can be used to define complex VSPs such as males between age 20 - 40 and females between age 15-30 (use a [ProductSplitter](#) to split subpopulations by sex and age, and then a [CombinedSplitter](#) to join several smaller VSPs together).
- Use a heterogeneous mating scheme that clones most individuals to the next generation (year) and produce offspring from the mating VSP.

Example [ageStructured](#) gives an example of the evolution of age-structured population.

- Information fields `ind_id`, `father_id` and `mother_id` and operators [IdTagger](#) and [PedigreeTagger](#) are used to track pedigree information during evolution.
- A [CloneMating](#) mating scheme is used to copy surviving individuals and a [RandomMating](#) mating scheme is used to produce offspring.
- [IdTagger](#) and [PedigreeTagger](#) are used in the `ops` parameter of [RandomMating](#) because only new offspring should have a new ID and record parental IDs. If you use these operators in the `duringOps` parameter of the `evolve` function, individuals copied by [CloneMating](#) will have a new ID, and a missing parental ID.
- The resulting population is age-structured so Pedigrees could be extracted from such a population.
- The penetrance function is age dependent. Because this penetrance function is applied to all individuals at each year and an individual will have the disease once he or she is affected, this penetrance function is more or less a hazard function.

Example: *Example of the evolution of age-structured population.*

```

>>> import simuPOP as sim
>>> import random
>>> N = 10000
>>> pop = sim.Population(N, loci=1, infoFields=['age', 'ind_id', 'father_id', 'mother_
↳ id'])
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', cutoff=[20, 50, 75]))
>>> def demoModel(gen, pop):
...     '''A demographic model that keep a constant supply of new individuals'''
...     # number of individuals that will die
...     sim.stat(pop, popSize=True, subPops=[(0,3)])
...     # individuals that will be kept, plus some new guys.

```

(continues on next page)

(continued from previous page)

```

...     return pop.popSize() - pop.dvars().popSize + N // 75
...
>>> def pene(geno, age, ind):
...     'Define an age-dependent penetrance function'
...     # this disease does not occur in children
...     if age < 16:
...         return 0
...     # if an individual is already affected, keep so
...     if ind.affected():
...         return 1
...     # the probability of getting disease increases with age
...     return (0., 0.001*age, 0.001*age)[sum(geno)]
...
>>> def outputstat(pop):
...     'Calculate and output statistics'
...     sim.stat(pop, popSize=True, numOfAffected=True,
...             subPops=[(0, sim.ALL_AVAIL)],
...             vars=['popSize_sp', 'propOfAffected_sp'])
...     for sp in range(3):
...         print('%s: %.3f%% (size %d)' % (pop.subPopName((0,sp)),
...             pop.dvars((0,sp)).propOfAffected * 100.,
...             pop.dvars((0,sp)).popSize))
...     #
...     return True
...
>>>
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         # random assign age
...         sim.InitInfo(lambda: random.randint(0, 75), infoFields='age'),
...         # random genotype
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         # assign an unique ID to everyone.
...         sim.IdTagger(),
...         sim.PyOutput('Prevalence of disease in each age group:\n'),
...     ],
...     # increase the age of everyone by 1 before mating.
...     preOps=sim.InfoExec('age += 1'),
...     matingScheme=sim.HeteroMating([
...         # all individuals with age < 75 will be kept. Note that
...         # CloneMating will keep individual sex, affection status and all
...         # information fields (by default).
...         sim.CloneMating(subPops=[(0,0), (0,1), (0,2)], weight=-1),
...         # only individuals with age between 20 and 50 will mate and produce
...         # offspring. The age of offspring will be zero.
...         sim.RandomMating(ops=[
...             sim.IdTagger(), # give new born an ID
...             sim.PedigreeTagger(), # track parents of each individual
...             sim.MendelianGenoTransmitter(), # transmit genotype
...         ]),
...         numOffspring=(sim.UNIFORM_DISTRIBUTION, 1, 3),
...         subPops=[(0,1)],),
...         subPopSize=demoModel),
...     # number of individuals?
...     postOps=[
...         sim.PyPenetrance(func=pene, loci=0),

```

(continues on next page)

(continued from previous page)

```

...     sim.PyOperator(func=outputstat, step=20)
...     ],
...     gen = 200
... )
Prevalence of disease in each age group:
age < 20: 0.578% (size 2596)
20 <= age < 50: 2.649% (size 4002)
50 <= age < 75: 4.217% (size 3249)
age < 20: 0.526% (size 2660)
20 <= age < 50: 27.627% (size 3931)
50 <= age < 75: 50.317% (size 3313)
age < 20: 0.489% (size 2660)
20 <= age < 50: 28.470% (size 3927)
50 <= age < 75: 61.757% (size 3347)
age < 20: 0.639% (size 2660)
20 <= age < 50: 29.449% (size 3990)
50 <= age < 75: 62.384% (size 3246)
age < 20: 0.526% (size 2660)
20 <= age < 50: 27.694% (size 3990)
50 <= age < 75: 64.030% (size 3325)
age < 20: 0.865% (size 2660)
20 <= age < 50: 28.070% (size 3990)
50 <= age < 75: 60.782% (size 3325)
age < 20: 0.489% (size 2660)
20 <= age < 50: 29.624% (size 3990)
50 <= age < 75: 60.812% (size 3325)
age < 20: 0.526% (size 2660)
20 <= age < 50: 29.273% (size 3990)
50 <= age < 75: 61.714% (size 3325)
age < 20: 0.789% (size 2660)
20 <= age < 50: 27.769% (size 3990)
50 <= age < 75: 61.233% (size 3325)
age < 20: 0.639% (size 2660)
20 <= age < 50: 29.073% (size 3990)
50 <= age < 75: 59.669% (size 3325)
200
>>>
>>> # draw two Pedigrees from the last age-structured population
>>> from simuPOP import sampling
>>> sample = sampling.drawNuclearFamilySample(pop, families=2, numOffspring=(2,3),
...     affectedParents=(1,2), affectedOffspring=(1,3))
>>> sim.dump(sample)
Ploidy: 2 (diploid)
Chromosomes:
1: (AUTOSOME, 1 loci)
(1)
Information fields:
age ind_id father_id mother_id
population size: 8 (1 subpopulations with 8 Individuals)
Number of ancestral populations: 0

SubPopulation 0 (), 8 Individuals:
0: MA 1 | 0 | 37 31578 27047 27596
1: MU 1 | 0 | 29 32638 29986 29012
2: MA 1 | 0 | 37 31579 27047 27596
3: FA 1 | 0 | 57 29012 25317 22955
4: MU 0 | 0 | 49 29986 27087 25888

```

(continues on next page)

(continued from previous page)

```

5: FA 1 | 1 | 67 27596 24124 24202
6: FA 1 | 0 | 29 32637 29986 29012
7: MA 1 | 0 | 71 27047 23653 20932

>>>

now exiting runScriptInteractively...
```

[Download ageStructured.py](#)

6.5 Tracing allelic lineage *

Lineage of alleles consists of information such as the distribution of alleles (how many people carry this allele, and the relationship between carriers) and age of alleles (when the alleles were introduced to the population). These information are important for the study of evolutionary history of mutants. They are not readily available for normal simulations, and even if you can track the generations when mutants are introduced, alleles in the present generation that are of the same type (Identity by Stat, IBS) do not necessarily have the same ancestral origin (Identity by Decent, IBD).

The lineage modules of simuPOP provides facilities to track allelic lineage. More specifically,

- Each allele is associated with an integer number (an allelic lineage) that identifies the origin, or the source of the allele.
- The lineage of each allele is transmitted along with the allele during evolution. New alleles will be introduced with their own lineage, even if they share the same states with existing alleles.
- Origin of alleles can be accessed using member functions of the *Individual* and *Population* classes.

Example *geneticContribution* demonstrates how to determine the contribution of genetic information from each ancestor. For this simulation, the alleles of each ancestor are associated with individual-specific numbers. During evolution, some alleles might get lost, some are copied, and pieces of chromosomes are mixed due to genetic recombination. At the end of simulation, the average number of ‘contributors’ of genetic information to each individual is calculated, as well as the percent of genetic information from each ancestor. Although this particular simulation can be mimicked using pure- genotype simulations by using special alleles for each ancestor, the combined information regarding the state and origin of each allele will be very useful for genetic studies that involve IBD and IBS.

Example: *Contribution of genetic information from ancestors*

```

>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage', quiet=True)
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[10]*4)
>>>
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.25]*4),
...         sim.InitLineage(range(1000), mode=sim.PER_INDIVIDUAL),
...     ],
...     matingScheme=sim.RandomMating(ops=sim.Recombinator(rates=0.001)),
...     gen = 100
... )
100
>>> # average number of 'contributors'
```

(continues on next page)

(continued from previous page)

```

>>> num_contributors = [len(set(ind.lineage())) for ind in pop.individuals()]
>>> print('Average number of contributors is %.2f' % (sum(num_contributors) /
↳float(pop.popSize())))
Average number of contributors is 13.98
>>> # percent of genetic information from each ancestor (baseline is 1/1000)
>>> lineage = pop.lineage()
>>> lin_perc = [lineage.count(x)/float(len(lineage)) for x in range(1000)]
>>> # how many of ancestors do not have any allele left?
>>> print('Number of ancestors with no allele left: %d' % lin_perc.count(0.))
Number of ancestors with no allele left: 817
>>> # top five contributors
>>> lin_perc.sort()
>>> lin_perc.reverse()
>>> print('Top contributors (started with 0.001): %.5f %.5f %.5f' % (lin_perc[0], lin_
↳perc[1], lin_perc[2]))
Top contributors (started with 0.001): 0.03474 0.03058 0.02475

now exiting runScriptInteractively...

```

Download geneticContribution.py

Example *geneticContribution* uses operator *InitLineage* to explicitly assign lineage to alleles of each individual. You can also track the fate of finer genetic pieces by assigning different lineage values to chromosomes, or each loci using different mode. This operator can also assign lineage of alleles to an ID stored in an information field, which is usually *ind_id*, a field used by operators such as *IdTagger* and *PedigreeTagger* to assign and trace the pedigree (parentship) information during evolution. More interesting, when such a field is present, mutation operators will assign the IDs of recipients of mutants as the lineage of these mutants. This makes it possible to track the origin of mutants. Moreover, when a mode *FROM_INFO_SIGNED* is used, additional ploidy information will be tagged to lineage values (negative values for mutants on the second homologous copy of chromosomes) so that you can track the inheritance of haplotypes.

To make use of these features, it is important to assign IDs to individuals before these operators are applied. Example *ageOfMutants* demonstrates how to use the lineage information to determine the age of mutants. This example evolves a constant population of size 10,000. An *IdTagger* is used before *InitGenotype* so individual IDs will be assigned as allelic lineages. Because all offspring get their own IDs during evolution, the IDs of individuals are assigned to mutants as their lineages, and can be used to determine the age of these mutants. This is pretty easy to do in this example because of constant population size. For more complex demographic models, you might have to record the minimal and maximum IDs of each generation in order to determine the age of mutants.

Example: Distribution of age of mutants

```

>>> import simuOpt
>>> simuOpt.setOptions(alleleType='lineage', quiet=True)
>>> import simuPOP as sim
>>> pop = sim.Population(size=10000, loci=[10]*10, infoFields='ind_id')
>>> # just to make sure IDs starts from 1
>>> sim.IdTagger().reset(1)
>>> pop.evolve(
...     initOps = [
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.2, 0.3, 0.4, 0.1]),
...         sim.IdTagger(),
...         sim.InitLineage(mode=sim.FROM_INFO),
...     ],
...     # an extremely high mutation rate, just for demonstration
...     preOps = sim.AcgtMutator(rate=0.01, model='JC69'),

```

(continues on next page)

(continued from previous page)

```

...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.IdTagger(),
...             sim.MendelianGenoTransmitter(),
...         ]
...     ),
...     gen = 10
... )
10
>>> lin = pop.lineage()
>>> # Number of alleles from each generation
>>> for gen in range(10):
...     id_start = gen*10000 + 1
...     id_end = (gen+1)*10000
...     num_mut = len([x for x in lin if x >= id_start and x <= id_end])
...     print('Gen %d: %5.2f %%' % (gen, num_mut / (2*10000*100.) * 100))
...
Gen 0: 93.40 %
Gen 1: 0.72 %
Gen 2: 0.71 %
Gen 3: 0.70 %
Gen 4: 0.74 %
Gen 5: 0.76 %
Gen 6: 0.73 %
Gen 7: 0.74 %
Gen 8: 0.75 %
Gen 9: 0.75 %

now exiting runScriptInteractively...

```

[Download ageOfMutants.py](#)

6.6 Pedigrees

6.6.1 Create a pedigree object

A *Pedigree* object is basically a static population object that is used to track relationship between individuals. A unique ID is required for all individuals so that individuals could be identified easily using their IDs. Individuals in a pedigree usually have one or two information fields to record the IDs of their parents. Operators *IdTagger* and *PedigreeTagger* are usually used to maintain these information fields which are, although customizable, almost always `ind_id`, `father_id` and `mother_id`. After pedigrees are identified, population operations could be applied, for example, to extracted identified pedigrees from an existing population. This is basically how module *simuPOP.sampling* works.

A new pedigree can be created from a population object with an ID field (default to `ind_id`), and two optional parental ID fields (default to `father_id` and `mother_id`). For example,

```
ped = Pedigree(pop, infoFields=ALL_AVAIL)
```

will create a pedigree object from population `pop` with information fields `ind_id`, `father_id` and `mother_id`, copying all available information fields. The ID field should have a unique ID for each individual and the parental ID fields should record the ID of his or her parents. Genotype information and additional information fields can be copied to a pedigree object if needed. The population object is unchanged.

Another method is to directly convert a population object to a pedigree object, using member function `asPedigree` of a population class. For example,

```
pop.asPedigree()
```

will convert the existing population to a pedigree object. Object `pop` can then be able to call all pedigree member functions. Once your task is done, you can convert the object back to a population using the `Pedigree.asPopulation()` member function of the object.

A pedigree object can also be created from a file saved by function `Pedigree.save()` or operator `PedigreeTagger` using function `loadPedigree`. Please refer to section *save and load pedigrees* in details.

6.6.2 Locate close and remote relatives of each individual

A pedigree object provides several functions for you to identify spouse, sibling and more distant relatives of each individual. The results are stored to additional information fields of each individual. For example, if you would like to know the offspring of all individuals, you can call function `Pedigree.locateRelatives` as follows:

```
offFields = ['off1', 'off2', 'off3']
ped.addInfoFields(offFields)
ped.locateRelatives(OFFSPRING, resultFields=offFields)
```

This function will locate up to 3 (determined by the length of `resultFields`) offspring of each individual and put their IDs in specified informaton fields. This function allows you to identify spouses (it is common to have multiple spouses when random mating is used), outbred spouse (exclude spouses who share at least one of the parents), offspring (all offspring) and common offspring with a specified spouse, siblings (share at least one parent) and full siblings (share two parents). It also allows you to limit the result by sex and affection status (e.g. find only affected female offspring).

More distant relationship can be derived from these relationship using function `Pedigree.traceRelatives`. This function accepts a path of information fields and follows the path to identify relatives. For example

```
sibFields = ['sib1', 'sib2']
offFields = ['off1', 'off2', 'off3']
cousinFields = ['cousin1', 'cousin2', 'cousin3']
ped.addInfoFields(sibFields + offFields + cousinFields)
ped.locateRelatives(FULLSIBLING, resultFields=sibFields)
ped.locateRelatives(OFFSPRING, resultFields=offFields)
ped.traceRelatives(['father_id', 'mother_id'], sibFields, offFields,
    sex=[ANY_SEX, MALE_ONLY, FEMALE_ONLY],
    resultField=cousinFields)
```

would first identify full siblings and offspring of all individuals and then locate father or mother's male sibling's daughters. As you can imagine, this function can be used to track very complicated relationships.

This function also provides a function for you to identify individuals with specified relatives. Example `locateRelative` gives an example how to locate a grandfather with at least five grandchildren. With such information, functions such as `Population.extractIndividuals()` could be used to extract Pedigrees from a population. This is basically how `simuPOP.sampling` module works.

Example: *Locate close and distant relatives of individuals*

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, ancGen=2, infoFields=['ind_id', 'father_id', 'mother_id',
↳ ''])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
```

(continues on next page)

(continued from previous page)

```

...     sim.IdTagger(),
... ],
... matingScheme=sim.RandomMating(
...     numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...     ops=[
...         sim.MendelianGenoTransmitter(),
...         sim.IdTagger(),
...         sim.PedigreeTagger()
...     ],
... ),
...     gen = 5
... )
5
>>> ped = sim.Pedigree(pop)
>>> offFields = ['off%d' % x for x in range(4)]
>>> grandOffFields = ['grandOff%d' % x for x in range(5)]
>>> ped.addInfoFields(['spouse'] + offFields + grandOffFields)
>>> # only look spouse for fathers...
>>> ped.locateRelatives(sim.OUTBRED_SPOUSE, ['spouse'], sex=sim.FEMALE_ONLY)
>>> ped.locateRelatives(sim.COMMON_OFFSPRING, ['spouse'] + offFields)
>>> # trace offspring of offspring
>>> ped.traceRelatives([offFields, offFields], resultFields=grandOffFields)
True
>>> #
>>> IDs = ped.individualsWithRelatives(grandOffFields)
>>> # check on ID.
>>> grandFather = IDs[0]
>>> grandMother = ped.indByID(grandFather).spouse
>>> # some ID might be invalid.
>>> children = [ped.indByID(grandFather).info(x) for x in offFields]
>>> childrenSpouse = [ped.indByID(x).spouse for x in children if x >= 1]
>>> childrenParents = [ped.indByID(x).father_id for x in children if x >= 1] \
...     + [ped.indByID(x).mother_id for x in children if x >= 1]
>>> grandChildren = [ped.indByID(grandFather).info(x) for x in grandOffFields]
>>> grandChildrenParents = [ped.indByID(x).father_id for x in grandChildren if x >=
↳1] \
...     + [ped.indByID(x).mother_id for x in grandChildren if x >= 1]
>>>
>>> def idString(IDs):
...     uniqueIDs = list(set(IDs))
...     uniqueIDs.sort()
...     return ', '.join(['%d' % x for x in uniqueIDs if x >= 1])
...
>>> print('GrandParents: %d, %d
... Children: %s
... Spouses of children: %s
... Parents of children: %s
... GrandChildren: %s
... Parents of grandChildren: %s' % \
...     (grandFather, grandMother, idString(children), idString(childrenSpouse),
...     idString(childrenParents), idString(grandChildren),
↳idString(grandChildrenParents)))
GrandParents: 3040, 3847
Children: 4078, 4079, 4080
Spouses of children: 4446, 4797
Parents of children: 3040, 3847
GrandChildren: 5188, 5189, 5879, 5880, 5881

```

(continues on next page)

(continued from previous page)

```

Parents of grandchildren: 4078, 4079, 4446, 4797
>>>
>>> # let us look at the structure of this complete pedigree using another method
>>> famSz = ped.identifyFamilies()
>>> # it is amazing that there is a huge family that connects almost everyone
>>> len(famSz), max(famSz)
(533, 2383)
>>> # if we only look at the last two generations, things are much better
>>> ped.addInfoFields('ped_id')
>>> famSz = ped.identifyFamilies(pedField='ped_id', ancGens=[0,1])
>>> len(famSz), max(famSz)
(664, 114)

now exiting runScriptInteractively...

```

[Download locateRelative.py](#)

6.6.3 Identify pedigrees (related individuals)

The *Pedigree* class provides some other functions that allows you to identify related individuals. For example,

- Function *Pedigree.identifyAncestors* identifies all ancestors of specified individuals or all individuals at the present generation. In a diploid population when there is only one parent, you can see that only a small portion of ancestors have offspring in the last generation.
- Function *Pedigree.identifyOffspring* identifies all offspring of specified individuals across multiple generations.
- Function *Pedigree.identifyFamilies* groups all related individuals into families and assign a family ID to all family members. You might be surprised by how large this kind of family can be when parents are allowed to have multiple spouses.

All these functions support parameters *subPops* and *ancGens* so that you can limit your search in specific sub-populations and ancestral generations. For example, you can limit your search to all male individuals to find out someone's male offspring. Example *locateFamilies* demonstrates how to use these functions to analyze the structure of a complete pedigree.

Example: *Identify all ancestors*

```

>>> import simuPOP as sim
>>> pop = sim.Population(1000, ancGen=-1, infoFields=['ind_id', 'father_id', 'mother_
↳ id'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...     ],
...     matingScheme=sim.RandomMating(
...         numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.IdTagger(),
...             sim.PedigreeTagger()
...         ],
...     ),
...     gen = 19

```

(continues on next page)

(continued from previous page)

```

... )
19
>>> # we now have the complete pedigree of 20 generations
>>> pop.asPedigree()
>>> # total number of individuals should be 20 * 1000
>>> # how many families do we have?
>>> fam = pop.identifyFamilies()
>>> len(fam)
525
>>> # but how many families with more than 1 individual?
>>> # The rest of them must be in the initial generation
>>> len([x for x in fam if x > 1])
18
>>> # let us look backward. allAnc are the ancestors who have offspring in the
>>> # last generation. You can see this is a small number compared the number of
>>> # ancestors.
>>> allAnc = pop.identifyAncestors()
>>> len(allAnc)
8614

now exiting runScriptInteractively...

```

[Download locateFamilies.py](#)

6.6.4 Save and load pedigrees

A complete pedigree, including ID, sex and affection status of each individual, IDs of their parents, and optionally values of some information fields and genotypes at some loci could be saved to a file, and be loaded using function `loadPedigree`. The loaded pedigree could be analyzed using pedigree functions, or be used to direct the evolution of another evolutionary process using a pedigree mating scheme.

A pedigree could be saved in two ways. In the first method, a pedigree could be created using the methods described above and be saved using function `Pedigree.save()`. However, if the population is large, recording all ancestral generations may not be feasible. If this is the case, you can use a `PedigreeTagger` operator to save individual information during the evolution. If you do not care about details of the top-most ancestral generation, a `PedigreeTagger` used in a mating scheme should be enough to record pedigree information of all offspring. Individual in the top-most generation who have offspring in the next generation will be constructed in `loadPedigree`. If you would like to include detailed information about all individuals in the top-most ancestral generation, you can use a `PedigreeTagger` in the `initOps` parameter of the `Simulator.evolve()` or `Population.evolve()` function.

Example `saveLoadPedigree` demonstrates how to use these functions to analyze the structure of a complete pedigree.

Example: *Save and load a complete pedigree*

```

>>> import simuPOP as sim
>>> pop = sim.Population(4, loci=1, infoFields=['ind_id', 'father_id', 'mother_id'],
...     ancGen=-1)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.IdTagger(),
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         sim.PedigreeTagger(output='>>pedigree.ped', outputLoci=0)
...     ],

```

(continues on next page)

(continued from previous page)

```

...     matingScheme=sim.RandomMating(
...         ops=[
...             sim.MendelianGenoTransmitter(),
...             sim.IdTagger(),
...             sim.PedigreeTagger(output='>>pedigree.ped', outputLoci=0)
...         ],
...     ),
...     gen = 2
... )
2
>>> #
>>> print(open('pedigree.ped').read())
1 0 0 F U 0 0
2 0 0 F U 0 1
3 0 0 M U 1 1
4 0 0 M U 1 1
5 4 1 M U 0 1
6 4 2 F U 1 1
7 3 2 F U 0 1
8 3 2 M U 1 1
9 8 7 F U 1 1
10 5 6 M U 1 1
11 5 6 M U 1 1
12 5 7 F U 0 1

>>> pop.asPedigree()
>>> pop.save('pedigree1.ped', loci=0)
>>> print(open('pedigree1.ped').read())
1 0 0 F U 0 0
2 0 0 F U 0 1
3 0 0 M U 1 1
4 0 0 M U 1 1
5 4 1 M U 0 1
6 4 2 F U 1 1
7 3 2 F U 0 1
8 3 2 M U 1 1
9 8 7 F U 1 1
10 5 6 M U 1 1
11 5 6 M U 1 1
12 5 7 F U 0 1

>>> #
>>> ped = sim.loadPedigree('pedigree1.ped')
>>> sim.dump(ped, ancGens=range(3))
Ploidy: 2 (diploid)
Chromosomes:
1:  (AUTOSOME, 1 loci)
    (1)
Information fields:
ind_id father_id mother_id
population size: 4 (1 subpopulations with 4 Individuals)
Number of ancestral populations: 2

SubPopulation 0 (), 4 Individuals:
  0: FU 1 | 1 | 9 8 7
  1: MU 1 | 1 | 10 5 6
  2: MU 1 | 1 | 11 5 6

```

(continues on next page)

(continued from previous page)

```

3: FU 0 | 1 | 12 5 7

Ancestral population 1
SubPopulation 0 (), 4 Individuals:
0: MU 0 | 1 | 5 4 1
1: FU 1 | 1 | 6 4 2
2: FU 0 | 1 | 7 3 2
3: MU 1 | 1 | 8 3 2

Ancestral population 2
SubPopulation 0 (), 4 Individuals:
0: FU 0 | 0 | 1 0 0
1: FU 0 | 1 | 2 0 0
2: MU 1 | 1 | 3 0 0
3: MU 1 | 1 | 4 0 0

```

[Download saveLoadPedigree.py](#)

6.7 Evolve a population following a specified pedigree structure **

There are some applications where you would like to repeat the same evolutionary process repeatedly using the same pedigree structure. For example, a gene-dropping simulation method basically initialize leaves of a pedigree with random genotypes and pass the genotypes along the pedigree according to Mendelian laws. This can be done in simuPOP using a pedigree mating scheme.

A pedigree mating scheme *PedigreeMating* evolves a population following an existing pedigree structure. If the *Pedigree* object has N ancestral generations and a present generation, it can be used to evolve a population for N generations, starting from the topmost ancestral generation. At the k -th generation, this mating scheme produces an offspring generation according to subpopulation structure of the $N-k-1$ ancestral generation in the pedigree object (e.g. producing the offspring population of generation 0 according to the $N-1$ ancestral generation of the pedigree object). For each offspring, this mating scheme copies individual ID and sex from the corresponding individual in the pedigree object. It then locates the parents of each offspring using their IDs in the pedigree object. A list of during mating operators are then used to transmit parental genotype to the offspring.

To use this mating scheme, you should

- Prepare a pedigree object with N ancestral generations (and a present generation). Parental information should be available at the present, parental, ..., and $N-1$ ancestral generations. This object could be created by evolving a population with `ancGen` set to -1 with parental information tracked by operators `idTagger()` and `pedigreeTagger()`.
- Prepare the population so that it contains individuals with IDs matching this generation, or at least individuals who have offspring in the next topmost ancestral generation. Because individuals in such a population will parent offsprings at the $N-1$ ancestral generation of the pedigree object, it is a good idea to assign `ind_id` using `ped.indInfo('father_id')` and `ped.infInfo('mother_id')` of the $N-1$ ancestral generation of `ped`.
- Evolve the population using a *PedigreeMating* mating scheme for N or less generations. Because parents are chosen by their IDs, subpopulation structure is ignored and migration will have no effect on the evolutionary process. No *IdTagger* should be used to assign IDs to offspring because re-labeling IDs will confuse this mating scheme. This mating scheme copies individual sex from pedigree individual to each offspring because individual sex may affect the way genotypes are transmitted (e.g. a *MendelianGenoTransmitter()* with sex chromosomes).

Example *pedigreeMating* demonstrates how to create a complete pedigree by evolving a population without genotype, and then replay the evolutionary process using another population.

Example: *Use a pedigree mating scheme to replay an evolutionary process.*

```
>>> import simuPOP as sim
>>> # create a population without any genotype
>>> from simuPOP.utils import migrSteppingStoneRates
>>> ped = sim.Population(size=[1000]*5, ancGen=-1,
...   infoFields=['ind_id', 'father_id', 'mother_id', 'migrate_to'])
>>> ped.evolve(
...   initOps=[
...     sim.InitSex(),
...     sim.IdTagger(),
...   ],
...   preOps=sim.Migrator(rate=migrSteppingStoneRates(0.1, 5)),
...   matingScheme=sim.RandomMating(
...     numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...     ops=[
...       # we do not even need a genotype transmitter...
...       sim.IdTagger(),
...       sim.PedigreeTagger(),
...     ],
...   ),
...   gen=100
... )
100
>>> # convert itself to a pedigree object
>>> ped.asPedigree()
>>> # we should have 100 ancestral generations
>>> N = ped.ancestralGens()
>>> # We should have 101 * 1000 * 5 individuals, but how many actually
>>> # contribute genotype to the last generation?
>>> anc = ped.identifyAncestors()
>>> len(anc)
205647
>>> # remove individuals who do not contribute genotype to the last generation
>>> allIDs = [x.ind_id for x in ped.allIndividuals()]
>>> removedIDs = list(set(allIDs) - set(anc))
>>> ped.removeIndividuals(IDs=removedIDs)
>>> # now create a top most population, but we do not need all of them
>>> # so we record only used individuals
>>> IDs = [x.ind_id for x in ped.allIndividuals(ancGens=N)]
>>> sex = [x.sex() for x in ped.allIndividuals(ancGens=N)]
>>> # create a population, this time with genotype. Note that we do not need
>>> # populaton structure because PedigreeMating disregard population structure.
>>> pop = sim.Population(size=len(IDs), loci=1000, infoFields='ind_id')
>>> # manually initialize ID and sex
>>> sim.initInfo(pop, IDs, infoFields='ind_id')
>>> sim.initSex(pop, sex=sex)
>>> pop.evolve(
...   initOps=sim.InitGenotype(freq=[0.4, 0.6]),
...   # we do not need migration, or set number of offspring,
...   # or demographic model, but we do need a genotype transmitter
...   matingScheme=sim.PedigreeMating(ped,
...     ops=sim.MendelianGenoTransmitter()),
...   gen=100
... )
100
```

(continues on next page)

(continued from previous page)

```

>>> # let us compare the pedigree and the population object
>>> print(ped.indInfo('ind_id')[:5])
(500001.0, 500002.0, 500003.0, 500004.0, 500005.0)
>>> print(pop.indInfo('ind_id')[:5])
(500001.0, 500002.0, 500003.0, 500004.0, 500005.0)
>>> print([ped.individual(x).sex() for x in range(5)])
[1, 2, 1, 1, 2]
>>> print([pop.individual(x).sex() for x in range(5)])
[1, 2, 1, 1, 2]
>>> print(ped.subPopSizes())
(663, 1254, 1213, 1230, 640)
>>> print(pop.subPopSizes())
(663, 1254, 1213, 1230, 640)

now exiting runScriptInteractively...

```

Download pedigreeMating.py

As long as unique IDs are used for individuals in different generations, the same technique could be used for overlapping generations as well. Even if some individuals are copied from generation to generation, separate IDs should be assigned to these individuals so that a pedigree could be correctly constructed. Because these individuals are copied from a single parent, the pedigree object will have mixed number of parents (some individuals have one parent, some have two). If *PedigreeTagger* operators are used to record parental information, such a pedigree could be loaded by function *loadPedigree*. Example *pedigreeMatingAgeStructured* evolves an age-structured population. Instead of saving all ancestral generations to a population object and convert it to a pedigree, this example saves the complete pedigree to file `structure.ped` and load the pedigree using function *loadPedigree*.

Example: *Replay an evolutionary process of an age-structured population*

```

>>> import simuPOP as sim
>>>
>>> import random
>>> N = 10000
>>> pop = sim.Population(N, infoFields=['age', 'ind_id', 'father_id', 'mother_id'])
>>> # we simulate age 0, 1, 2, 3
>>> pop.setVirtualSplitter(sim.InfoSplitter(field='age', values=[0, 1, 2, 3]))
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         # random assign age
...         sim.InitInfo(lambda: random.randint(0, 3), infoFields='age'),
...         # random genotype
...         sim.InitGenotype(freq=[0.5, 0.5]),
...         # assign an unique ID to everyone.
...         sim.IdTagger(),
...     ],
...     # increase the age of everyone by 1 before mating.
...     preOps=sim.InfoExec('age += 1'),
...     matingScheme=sim.HeteroMating([
...         # age 1, 2 will be copied
...         sim.CloneMating(
...             ops=[
...                 # This will set offspring ID
...                 sim.CloneGenoTransmitter(),
...                 # new ID for offspring in order to track pedigree
...                 sim.IdTagger(),
...                 # both offspring and parental IDs will be the same

```

(continues on next page)

(continued from previous page)

```

...         sim.PedigreeTagger(output='>>structured.ped'),
...     ],
...     subPops=[(0,1), (0,2)],
...     weight=-1
... ),
...     # age 2 produce offspring
...     sim.RandomMating(
...         ops=[
...             # new ID for offspring
...             sim.IdTagger(),
...             # record complete pedigree
...             sim.PedigreeTagger(output='>>structured.ped'),
...             sim.MendelianGenoTransmitter(), # transmit genotype
...         ],
...         subPops=[(0,2)]
...     )]
... ),
...     gen=20
... )
20
>>>
>>> # use a pedigree object recovered from a file saved by operator PedigreeTagger
>>> ped = sim.loadPedigree('structured.ped')
>>> # create a top most population, but we do not need all of them
>>> # so we record only used individuals
>>> IDs = [x.ind_id for x in ped.allIndividuals(ancGens=ped.ancestralGens())]
>>> sex = [x.sex() for x in ped.allIndividuals(ancGens=ped.ancestralGens())]
>>> # create a population, this time with genotype. Note that we do not need
>>> # populaton structure because PedigreeMating disregard population structure.
>>> pop = sim.Population(size=len(IDs), loci=1000, infoFields='ind_id')
>>> # manually initialize ID and sex
>>> sim.initInfo(pop, IDs, infoFields='ind_id')
>>> sim.initSex(pop, sex=sex)
>>> pop.evolve(
...     initOps=sim.InitGenotype(freq=[0.4, 0.6]),
...     # we do not need migration, or set number of offspring,
...     # or demographic model, but we do need a genotype transmitter
...     matingScheme=sim.PedigreeMating(ped,
...         ops=sim.IfElse(lambda mom: mom is None,
...             sim.CloneGenoTransmitter(),
...             sim.MendelianGenoTransmitter())
...     ),
...     gen=100
... )
20
>>> #
>>> print(pop.indInfo('ind_id')[:5])
(200001.0, 200002.0, 200003.0, 200004.0, 200005.0)
>>> print([pop.individual(x).sex() for x in range(5)])
[1, 2, 2, 1, 1]
>>> # The pedigree object does not have population structure
>>> print(pop.subPopSizes())
(10000,)

now exiting runScriptInteractively...
```

[Download pedigreeMatingAgeStructured.py](#)

The pedigree is then used to repeat the evolutionary process. However, because some individuals were produced sexually using *MendelianGenoTransmitter* and some were copied using *CloneGenoTransmitter*, an *IfElse* operator has to be used to transmit genotypes correctly. This example uses the function condition of the *IfElse* operator and makes use of the fact that parent `mom` will be `None` if an individual is copied from his or her father.

plainnat simuPOP

6.8 Simulation of mitochondrial DNAs (mtDNAs) *

Mitochondrial DNAs resides in human mitochondrion. A zygote inherits its organelles from the cytoplasm of the egg, and thus organelle inheritance is generally maternal. Whereas there is only one copy of a nuclear chromosome per gamete, there are many copies of an organellar chromosome, forming a population of identical organelle chromosomes that is transmitted to the offspring through the egg. Because these organellar chromosomes are identical, they are modelled in simuPOP as a single chromosome with type `MITOCHONDRIAL`. In order to simulate mitochondrial DNAs, it is important to remember:

- *MendelianGenoTransmitter* and *Recombinator* do not handle mitochondrial DNAs so you will have to explicitly use *MitochondrialGenoTransmitter* to transmit the mitochondrial DNAs from mother to offspring. Note that *CloneGenoTransmitter* is a special transmitter that will copy everything including sex, information fields to offspring.
- The *Stat* operator recognizes this chromosome type and will report allele, haplotype, and genotype counts, and other statistics correctly, although some diploid-specific statistics are not applicable.
- Natural selections on mtDNAs is usually performed using operator *MapSelector* where single alleles are assigned a fitness value. Operator *MaSelector* assumes two alleles and is not applicable.

Example *mitochondrial* demonstrates the use of a *Recombinator* to recombine an autosome and two sex chromosomes, and a *MitochondrialGenoTransmitter* to transmit mitochondrial chromosomes. Natural selection is applied to allele 3 at the 3rd locus on the mitochondrial DNA, whose frequency in the population decreases as a result.

Example: *Transmission of mitochondrial chromosomes*

```
>>> import simuPOP as sim
>>> pop = sim.Population(1000, loci=[5]*4,
... # one autosome, two sex chromosomes, and one mitochondrial chromosomes
... chromTypes=[sim.AUTOSOME, sim.CHROMOSOME_X, sim.CHROMOSOME_Y, sim.
↳MITOCHONDRIAL],
... infoFields=['fitness'])
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.25]*4)
...     ],
...     preOps=[
...         sim.MapSelector(loci=17, fitness={(0,): 1, (1,): 1, (2,): 1, (3,): 0.4})
...     ],
...     matingScheme=sim.RandomMating(ops= [
...         sim.Recombinator(rates=0.1),
...         sim.MitochondrialGenoTransmitter(),
...     ]),
...     postOps=[
...         sim.Stat(alleleFreq=17, step=10),
...         sim.PyEval(r'("%.2f %.2f %.2f %.2f\n" % (alleleNum[17][0], '
...             'alleleNum[17][1], alleleNum[17][2], alleleNum[17][3]))', step=10),
...     ],
...     gen = 100
```

(continues on next page)

(continued from previous page)

```

... )
1288.00 273.00 325.00 114.00
1384.00 245.00 371.00 0.00
1492.00 138.00 370.00 0.00
1461.00 69.00 470.00 0.00
1449.00 65.00 486.00 0.00
1536.00 17.00 447.00 0.00
1624.00 7.00 369.00 0.00
1538.00 0.00 462.00 0.00
1619.00 0.00 381.00 0.00
1623.00 0.00 377.00 0.00
100

now exiting runScriptInteractively...

```

Download mitochondrial.py

You might wonder how a mutation can change the allele of all organelles in the mitochondrion. This is generally believed to be done through natural drift during cytoplasmic segregation, which is not a mitotic process because it takes place in dividing asexual cells. Because only one mitochondrial chromosome is allowed in simuPOP, you will have to use customized chromosome types if you would like to simulate this process. Fortunately, operator *MitochondrialGenoTransmitter* can select random organelles from multiple customized chromosomes, if no chromosome of type MITOCHONDRIAL is present.

Example *mtDNA_evolve* demonstrates the fixation of mutant in cells with multiple organelles. Although mutations are introduced to only one of the organelles, after a number of cell divisions, the majority of the cells now have only one type of allele. This example uses a *RandomSelection* mating scheme to select cells randomly from the parental population. Because no sexual reproduction is involved, *MitochondrialGenoTransmitter* passes the parental genotype to offspring regardless of sex of parent. This example also demonstrates a disadvantage of using customized chromosomes in that you will have to calculate statistics by yourself because only you know the meaning of these chromosomes. In this example, a function is written to count the number of mutants in each cell (individual), and summarize the number of cells with 0, 1, 2, 3, 4, and 5 copies of the mutant.

Example: Evolution of multiple organelles in mitochondrion

```

>>> import simuPOP as sim
>>>
>>> def alleleCount(pop):
...     summary = [0]* 6
...     for ind in pop.individuals():
...         geno = ind.genotype(ploidy=0)
...         summary[geno[0] + geno[2] + geno[4] + geno[6] + geno[8]] += 1
...     print('%d %s' % (pop.dvars().gen, summary))
...     return True
...
>>> pop = sim.Population(1000, loci=[2]*5, chromTypes=[sim.CUSTOMIZED]*5)
>>> pop.evolve(
...     # every one has mtDNAs 10, 00, 00, 00, 00
...     initOps=[
...         sim.InitGenotype(haplotypes=[[1]+[0]*9]),
...     ],
...     # random select cells for cytoplasmic segregation
...     matingScheme=sim.RandomSelection(ops=[
...         sim.MitochondrialGenoTransmitter(),
...     ]),
...     postOps=sim.PyOperator(func=alleleCount, step=10),
...     gen = 51

```

(continues on next page)

(continued from previous page)

```
... )
0 [333, 408, 219, 38, 2, 0]
10 [806, 16, 14, 16, 11, 137]
20 [816, 1, 1, 3, 0, 179]
30 [833, 0, 0, 0, 0, 167]
40 [805, 0, 0, 0, 0, 195]
50 [849, 0, 0, 0, 0, 151]
51

now exiting runScriptInteractively...
```

[Download mtDNA_evolve.py](#)

7.1 Module `simuOpt` (function `simuOpt.setOptions`)

Module `simuOpt` handles options to specify which simuPOP module to load and how this module should be loaded, using function `simuOpt.setOptions` with parameters *alleleType* (short, long, or binary), *optimized* (standard or optimized), *gui* (whether or not use a graphical user interface and which graphical toolkit to use), *revision* (minimal required version/revision), *quiet* (with or without banner message, and *debug* (which debug code to turn on). These options have been discussed in Example *lst_Use_of_standard_module* and *lst_Use_of_optimized_module* and other related sections. Note that **most options can be set by environmental variables and command line options** which are sometimes more versatile to use.

7.2 Module `simuPOP.utils`

The `simuPOP.utils` module provides a few utility functions and classes. They do not belong to the simuPOP core but are distributed with simuPOP because they are frequently used and play an important role in some specialized simulation techniques. Please refer to the simuPOP online cookbook (<http://simupop.sourceforge.net/cookbook>) for more utility modules and functions.

7.2.1 Trajectory simulation (classes `Trajectory` and `TrajectorySimulator`)

A forward-time simulation, by its nature, is directly influenced by random genetic drift. Starting from the same parental generation, allele frequencies in the offspring generation would vary from simulation to simulation, with perhaps a predictable mean frequency which is determined by factors such as parental allele frequency, natural selection, mutation and migration.

Genetic drift is unavoidable and is in many cases the target of theoretical and simulation studies. However, in certain types of studies, there is often a need to control the frequencies of certain alleles in the present generation. For example, if we are studying a particular penetrance model with pre-specified frequencies of disease predisposing alleles, the simulated populations would better have consistent allele frequencies at the disease predisposing loci, and consequently consistent disease prevalence.

simuPOP provides a special offspring generator *ControlledOffspringGenerator* and an associated mating scheme called *ControlledRandomMating* that can be used to generate offspring generations conditioning on frequencies of one or more alleles. This offspring generator essentially uses a reject-sampling algorithm to select (or reject) offspring according to their genotypes at specified loci. A detailed description of this algorithm is given in Peng2007a.

The controlled random mating scheme accepts a user-defined trajectory function that tells the mating scheme the desired allele frequencies at each generation. Example *controlledOffGenerator* uses a manually defined function that raises the frequency of an allele steadily. However, given known demographic and genetic factors, **a trajectory should be simulated randomly so that it represents a random sample from all possible trajectories that match the allele frequency requirement.** If such a condition is met, the controlled evolutionary process can be considered as a random process conditioning on allele frequencies at the present generation. Please refer to Peng2007a for a detailed discussion about the theoretical requirements of a valid trajectory simulator.

The *simuUtil* module provides functions and classes that implement two trajectory simulation methods that can be used in different situations. The first class is *TrajectorySimulator* which takes a demographic model and a selection model as its input and simulates allele frequency trajectories using a forward or backward algorithm. The demographic model is given by parameter *N*, which can be a constant (e.g. *N*=1000) for constant population size, a list of subpopulation sizes (e.g. *N*=[1000, 2000]) for a structured population with constant size, or a demographic function that returns population or subpopulation sizes at each generation. In the last case, subpopulations can be split or merged with the constraint that subpopulations can be merged into one, from split from one population.

A fitness model specifies the fitness of genotypes at one or more loci using parameter *fitness*. It can be a list of three numbers (e.g. *fitness*=[1, 1.001, 1.003]), representing the fitness of genotype AA, Aa and aa at one or more loci; or different fitness for genotypes at each locus (e.g. *fitness*=[1, 1.001, 1.003, 1, 1, 1.002]), or for each combination of genotype (interaction). In the last case, values are needed for each genotype if there are loci. This trajectory simulator also accepts generation-specific fitness values by accepting a function that returns fitness values at each generation.

The simulator then simulates trajectories of allele frequencies and return them as objects of class *Trajectory*. This object can be used provide a trajectory function that can be used directly in a *ControlledRandomMating* mating scheme (function *func()*) or provide a list of *PointMutator* to introduce mutants at appropriate generations (function *mutators()*). If a simulation failed after specified number of attempts, a *None* object will be returned.

Forward-time trajectory simulations (function *simulateForwardTrajectory*)

A forward simulation starts from a specified generation with specified allele frequencies at one or more loci. The simulator simulates allele frequencies forward-in-time, until it reaches a specified ending generation. A trajectory object will be returned if the simulated allele frequencies fall into specified ranges. Example *forwardTrajectory* demonstrates how to use this simulation method to obtain and use a simulated trajectory, for two unlinked loci under different selection pressure.

Example: *Simulation and use of forward-time simulated trajectories.*

```
>>> import simuOpt
>>> simuOpt.setOptions(quiet=True)
>>> import simuPOP as sim
>>> from simuPOP.utils import Trajectory, simulateForwardTrajectory
>>>
>>> traj = simulateForwardTrajectory(N=[2000, 4000], fitness=[1, 0.99, 0.98],
...     beginGen=0, endGen=100, beginFreq=[0.2, 0.3],
...     endFreq=[[0.1, 0.11], [0.2, 0.21]])
>>> #
>>> #traj.plot('log/forwardTrajectory.png', set_ylim_top=0.5,
>>> #     plot_c_sp=['r', 'b'], set_title_label='Simulated Trajectory (forward-time)')
>>> pop = sim.Population(size=[2000, 4000], loci=10, infoFields='fitness')
```

(continues on next page)

(continued from previous page)

```

>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.8, 0.2], subPops=0),
...         sim.InitGenotype(freq=[0.7, 0.3], subPops=1),
...         sim.PyOutput('Sp0: loc2\tloc5\tSp1: loc2\tloc5\n'),
...     ],
...     matingScheme=sim.ControlledRandomMating(
...         ops=[sim.Recombinator(rates=0.01)],
...         loci=5, alleles=1, freqFunc=traj.func()),
...     postOps=[
...         sim.Stat(alleleFreq=[2, 5], vars=['alleleFreq_sp'], step=20),
...         sim.PyEval(r"%0.2f\t%0.2f\t%0.2f\t%0.2f\n" % (subPop[0]['alleleFreq'][2][1],
...             "subPop[0]['alleleFreq'][5][1], subPop[1]['alleleFreq'][2][1], "
...             "subPop[1]['alleleFreq'][5][1])", step=20)
...     ],
...     gen = 101
... )
Sp0: loc2    loc5    Sp1: loc2    loc5
0.19 0.20    0.30    0.29
0.20 0.20    0.29    0.27
0.20 0.14    0.28    0.27
0.17 0.13    0.27    0.26
0.14 0.13    0.31    0.23
0.13 0.10    0.27    0.20
101
now exiting runScriptInteractively...

```

[Download forwardTrajectory.py](#)

Figure [fig_forwardTrajectory](#) plots simulated trajectories of one locus in two subpopulations. The plot function uses either rpy or matplotlib as the underlying plotting library.

Figure: *Simulated trajectories of one locus in two subpopulations*

Backward-time trajectory simulations (function `simulateBackwardTrajectory`).

A backward simulation starts from specified frequencies at the present generation. In the single-allele case, the simulation goes backward-in-time until an allele gets lost. The length of such a trajectory is random, which is usually a desired property because the age of a mutant in the present generation is usually unknown and is assumed to be random.

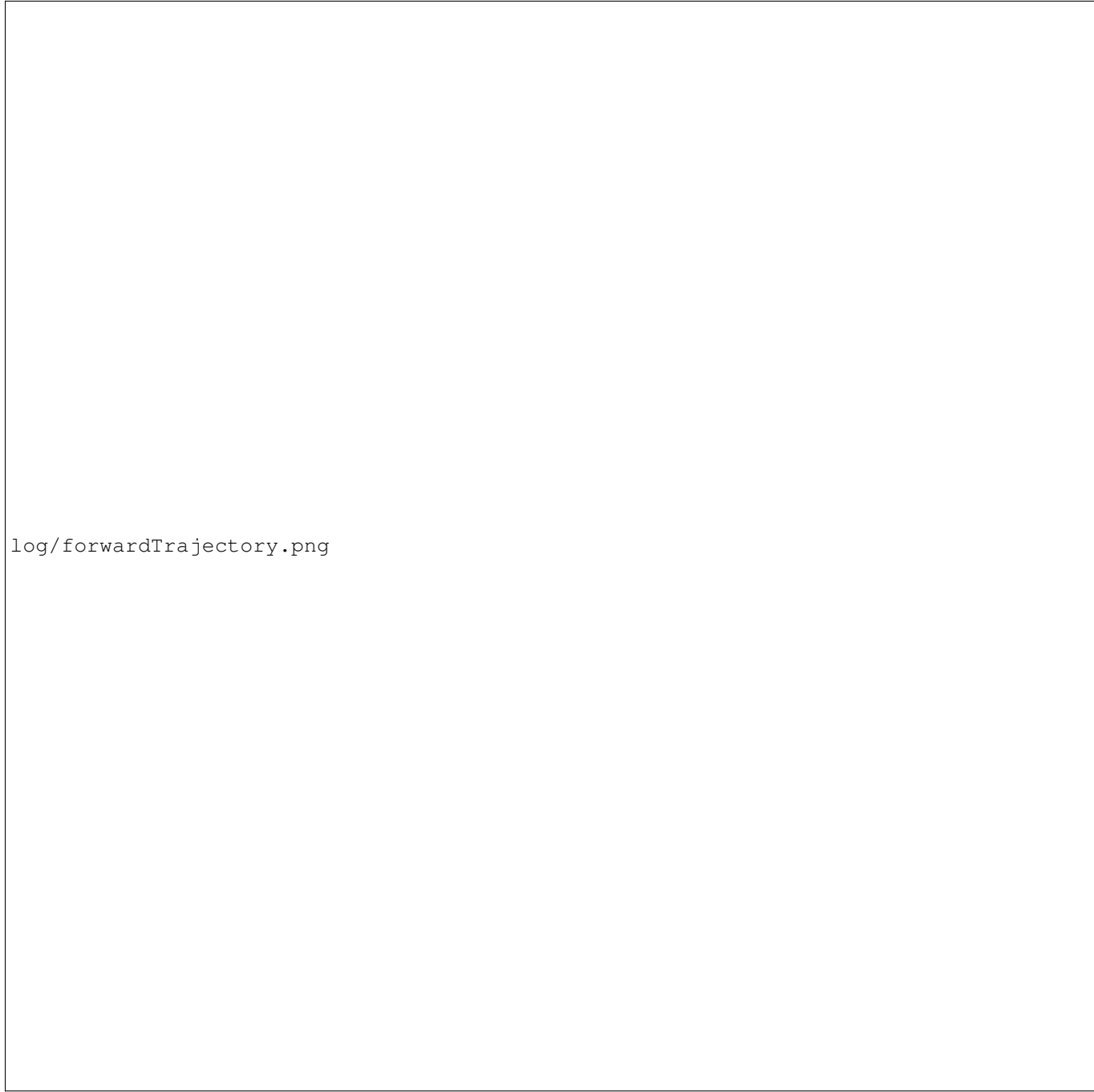
This trajectory simulation technique is usually used as follows:

1. Determine a demographic and a natural selection model using which a forward- time simulation will be performed.
2. Given current disease allele frequencies, simulate trajectories of allele frequencies at each DSL using a backward approach.
3. Evolve a population forward-in-time, using designed demographic and selection models. A [ControlledRandomMating](#) scheme instead of the usual [RandomMating](#) scheme should be used.

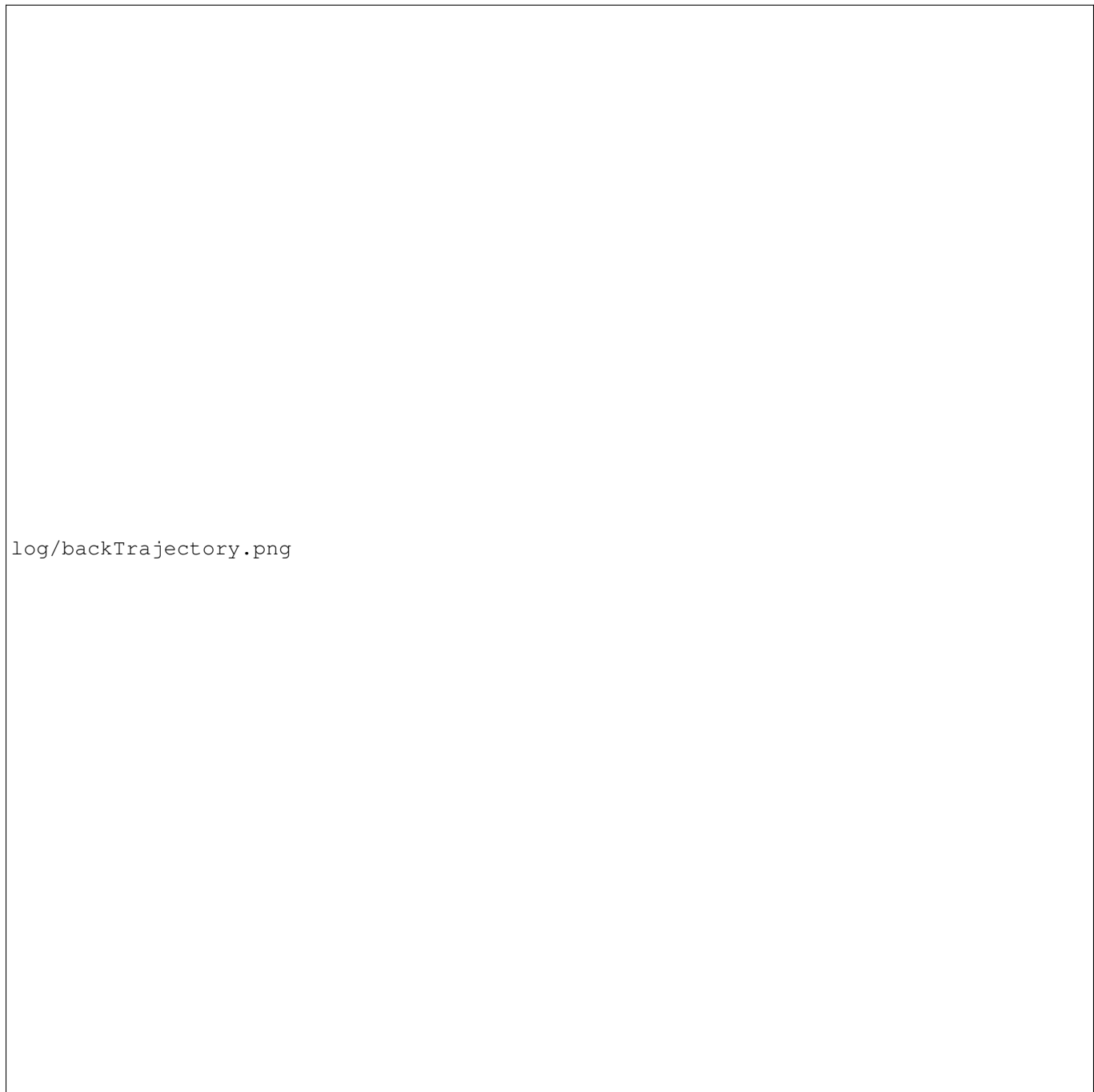
Figure [fig_backTrajectory](#) plots simulated trajectories of two unlinked loci.

Figure: *Simulated trajectories of two unlinked loci*

The trajectory is used in a [ControlledRandomMating](#) scheme in the following evolutionary scenario:



log/forwardTrajectory.png



log/backTrajectory.png

Example: *Simulation and use of backward-time simulated trajectories.*

```
>>> import simuPOP as sim
>>> from simuPOP.utils import Trajectory, simulateBackwardTrajectory
>>> from math import exp
>>> def Nt(gen):
...     'An exponential sim.Population growth demographic model.'
...     return int((5000) * exp(.00115 * gen))
...
>>> def fitness(gen, sp):
...     'Constant positive selection pressure.'
...     return [1, 1.01, 1.02]
...
>>> # simulate a trajectory backward in time, from generation 1000
>>> traj = simulateBackwardTrajectory(N=Nt, fitness=fitness, nLoci=2,
...     endGen=1000, endFreq=[0.1, 0.2])
>>> # matplotlib syntax
>>> #traj.plot('log/backTrajectory.png', set_ylim_top=0.3, set_ylim_bottom=0,
>>> #         plot_c_loc=['r', 'b'], set_title_label='Simulated Trajectory (backward-
↳time)')
>>>
>>> print('Trajectory simulated with length %s ' % len(traj.traj))
Trajectory simulated with length 834
>>> pop = sim.Population(size=Nt(0), loci=[1]*2)
>>> # save Trajectory function in the sim.population's local namespace
>>> # so that the sim.PyEval operator can access it.
>>> pop.dvars().traj = traj.func()
>>> pop.evolve(
...     initOps=[sim.InitSex()],
...     preOps=traj.mutators(loci=[0, 1]),
...     matingScheme=sim.ControlledRandomMating(loci=[0, 1], alleles=[1, 1],
...         subPopSize=Nt, freqFunc=traj.func()),
...     postOps=[
...         sim.Stat(alleleFreq=[0, 1], begin=500, step=100),
...         sim.PyEval(r"%4d: %.3f (exp: %.3f), %.3f (exp: %.3f)\n" % (gen,
↳alleleFreq[0][1],
...             "traj(gen)[0], alleleFreq[1][1], traj(gen)[1]"),
...             begin=500, step=100)
...     ],
...     gen=1001 # evolve 1001 generations to reach the end of generation 1000
... )
500: 0.013 (exp: 0.013), 0.000 (exp: 0.000)
600: 0.005 (exp: 0.005), 0.003 (exp: 0.003)
700: 0.011 (exp: 0.011), 0.008 (exp: 0.008)
800: 0.012 (exp: 0.013), 0.031 (exp: 0.031)
900: 0.037 (exp: 0.037), 0.092 (exp: 0.092)
1000: 0.101 (exp: 0.100), 0.200 (exp: 0.200)
1001
now exiting runScriptInteractively...
```

Download [backTrajectory.py](#)

7.2.2 Graphical or text-based progress bar (class `ProgressBar`)

If your simulation takes a while to finish, you could use a progress bar to indicate its progress. The `ProgressBar` class is provided for such a purpose. Basically, you create a `ProgressBar` project with intended total steps, and calls

its `update` member function with each progress. Depending on available graphical toolkit and the global or local GUI settings, a wxPython based dialog, a Tkinter based dialog, or a text-based dialog will be used. Example [ProgressBar](#) demonstrates how to use a text-based progress bar. If the progress bar is updated at each step (such as in this example), function `update()` can be called without parameter because it updates the progress bar at an increment of 1 in this case.

Example: *Using a text-based progress bar*

```
>>> import simuPOP as sim
>>> from simuPOP.utils import ProgressBar
>>> pop = sim.Population(10000, loci=[10], infoFields='index')
>>> prog = ProgressBar('Setting individual genotype...\n', pop.popSize(), gui=False)
Setting individual genotype...
>>> for idx in range(pop.popSize()):
...     # do something to each individual
...     pop.individual(idx).index = idx
...     # idx + 1 can be ignored in this case.
...     prog.update(idx + 1)
...
....1....2....3....4....5....6....7....8....9.... Done.

now exiting runScriptInteractively...
```

[Download ProgressBar.py](#)

7.2.3 Display population variables (function `viewVars`)

If a population has a large number of variables, or if you are not sure which variable to output, you could use function `viewVars` to view the population variables in a tree form. If wxPython is available, a dialog could be used to view the variables interactively. Example [viewVars](#) demonstrates how to use this function. The wxPython-based dialog is displayed in Figure [viewVars](#).

Example: *Using function `viewVars` to display population variables*

```
import simuPOP as sim
from simuPOP.utils import viewVars
pop = sim.Population([1000, 2000], loci=3)
sim.initGenotype(pop, freq=[0.2, 0.4, 0.4], loci=0)
sim.initGenotype(pop, freq=[0.2, 0.8], loci=2)
sim.stat(pop, genoFreq=[0, 1, 2], haploFreq=[0, 1, 2],
         alleleFreq=range(3),
         vars=['genoFreq', 'genoNum', 'haploFreq', 'alleleNum_sp'])
viewVars(pop.vars())
```

[Download viewVars.py](#)

Figure: *Using wxPython to display population variables*

7.2.4 Import simuPOP population from files in GENEPOP, PHYLIP and FSTAT formats (function `importPopulation`)

A function `importPopulation` is provided in the `simuPOP.utils` module to import populations from files in GENEPOP, PHYLIP and FSTAT formats. Because these formats do not support many of the features of a simuPOP population, this function can only import genotype and basic information of a population. Because formats GENEPOP and FSTAT formats uses allele 0 to indicate missing value, true alleles in these formats start at value 1. If you would



Users/bpeng1/simuPOP/simuPOP/doc/figures/viewVars.png

like to import alleles with starting value 0, you can use parameter `adjust=-1` to adjust imported values, if you data do not have any missing value.

7.2.5 Export simuPOP population to files in STRUCTURE, GENEPOP, FSTAT, Phylip, PED, MAP, MS, and CSV formats (function `export` and operator `Exporter`)

simuPOP uses a program-specific binary format to save and load populations but you can use the `export` function to export a simuPOP population in other formats if you would like to use other programs to analyze simulated populations. An operator `Exporter` is also provided so that you could export populations during evolution. Operator arameters such as `output`, `begin`, `end`, `step`, `at`, `reps`, and `subPops` are supported so that you could export subsets of individuals at multiple generations using different file names (e.g. `output='! '%d.ped' ' % gen'` to output to different files at different generations).

Commonly used population genetics file formats such as GENEPOP, FSTAT, Phylip, MS, and STRUCTURE are supported. Because these formats cannot store all information in a simuPOP population, export and import operations can lose information. Also, because the processing application have different assumptions, some conversion of genotypes might be needed. For example, because GENEPOP uses allele 0 as missing genotype, function `export(format='genepop')` accepts a parameter `adjust` with default value 1 to export alleles 0, 1 etc to 1, 2, The same applies to function `importPopulation` where some file formats accepts a parameter `adjust` (with default value 1) to adjust allele values. Please refer to the simuPOP reference manual for a detailed list of acceptable parameters for each format.

Example *importExport* demonstrates how to import and export a population in formats FSTAT and STRUCTURE. For the FSTAT format, because the population is exported with allele values shifted by 1, the imported population has different alleles than the original population. This can be fixed by adding parameter `adjust=-1` to the `importPopulation` function.

Example: *Save and load a population*

```
>>> import simuPOP as sim
>>> from simuPOP.utils import importPopulation, export
>>> pop = sim.Population([2,4], loci=5, lociNames=['a1', 'a2', 'a3', 'a4', 'a5'],
...     infoFields='BMI')
>>> sim.initGenotype(pop, freq=[0.3, 0.5, 0.2])
>>> sim.initSex(pop)
>>> sim.initInfo(pop, [20, 30, 40, 50, 30, 25], infoFields='BMI')
>>> export(pop, format='fstat', output='fstat.txt')
Exporting....1....2....3....4....5....6....7....8....9.... Done.
>>> print(open('fstat.txt').read())
2 5 3 1
a1
a2
a3
a4
a5
1 21 21 23 12 12
1 22 23 22 22 21
2 31 21 22 11 13
2 22 22 33 23 21
2 22 32 33 22 21
2 33 33 22 21 32

>>> export(pop, format='structure', phenotype='BMI', output='stru.txt')
Exporting....1....2....3....4....5....6....7....8....9.... Done.
>>> print(open('stru.txt').read())
```

(continues on next page)

(continued from previous page)

```

a1    a2    a3    a4    a5
-1    1.0    1.0    1.0    1.0
1     1     20     1     1     1     0     0
1     1     20     0     0     2     1     1
2     1     30     1     1     1     1     1
2     1     30     1     2     1     1     0
1     2     40     2     1     1     0     0
1     2     40     0     0     1     0     2
2     2     50     1     1     2     1     1
2     2     50     1     1     2     2     0
3     2     30     1     2     2     1     1
3     2     30     1     1     2     1     0
4     2     25     2     2     1     1     2
4     2     25     2     2     1     0     1

>>> pop1 = importPopulation(format='fstat', filename='fstat.txt')
>>> sim.dump(pop1)
Ploidy: 2 (diploid)
Chromosomes:
1: (AUTOSOME, 5 loci)
   a1 (1), a2 (2), a3 (3), a4 (4), a5 (5)
population size: 6 (2 subpopulations with 2 (1), 4 (2) Individuals)
Number of ancestral populations: 0

SubPopulation 0 (1), 2 Individuals:
  0: MU 22211 | 11322
  1: MU 22222 | 23221
SubPopulation 1 (2), 4 Individuals:
  2: MU 32211 | 11213
  3: MU 22322 | 22331
  4: MU 23322 | 22321
  5: MU 33223 | 33212

now exiting runScriptInteractively...
```

Download `importExport.py`

Because coalescent simulations are increasingly used to generate initial populations in equilibrium stats, importing data in MS format is very useful. Because MS only simulates haploid sequences with genotype only at segregating sites, you might have to simulate an even number of sequences and use option `ploidy=2` to import the simulated data as a haploid population. In addition, a parameter `mergeBy` is provided to import multiple replicates as multiple subpopulations or chromosomes. This corresponds to the `splitBy` parameter when you export your data in MS format. Example *importMS* demonstrates how to use these parameters.

Example: Export and import in MS format

```

>>> import simuPOP as sim
>>> from simuPOP.utils import importPopulation, export
>>> pop = sim.Population([20,20], loci=[10, 10])
>>> # simulate a population but mutate only a subset of loci
>>> pop.evolve(
...     preOps=[
...         sim.InitSex(),
...         sim.SNPMutator(u=0.1, v=0.01, loci=range(5, 17))
...     ],
...     matingScheme=sim.RandomMating(),
```

(continues on next page)

(continued from previous page)

```

...     gen=100
... )
100
>>> # export first chromosome, all individuals
>>> export(pop, format='ms', output='ms.txt')
Exporting....1....2....3....4....5....6....7....8....9.... Done.
>>> # export first chromosome, subpops as replicates
>>> export(pop, format='ms', output='ms_subPop.txt', splitBy='subPop')
Exporting....1....2....3....4....5....6....7....8....9.... Done.
>>> # export all chromosomes, but limit to all males in subPop 1
>>> pop.setVirtualSplitter(sim.SexSplitter())
>>> export(pop, format='ms', output='ms_chrom.txt', splitBy='chrom', subPops=[(1,0)])
Exporting....1....2....3....4....5....6....7....8....9.... Done.
>>> #
>>> print(open('ms_chrom.txt').read())
simuPOP_export 20 2
30164 48394 29292

//
segsites: 5
positions: 6.0 7.0 8.0 9.0 10.0
11110
11111
11110
11111
11011
11111
01111
10111
11111
11111
01111
01111
11011
11111
01111
11011
11101
10111
11111
11111

//
segsites: 7
positions: 1.0 2.0 3.0 4.0 5.0 6.0 7.0
1101111
1110011
1101110
1111111
0111110
1111111
1110001
1111111
0111110
1111111
1111111
1111111

```

(continues on next page)

(continued from previous page)

```

11111111
10111111
11111111
11111111
10111111
11111111
11111111
10111111

>>> # import as haploid sequence
>>> pop = importPopulation(format='ms', filename='ms.txt')
>>> # import as diploid
>>> pop = importPopulation(format='ms', filename='ms.txt', ploidy=2)
>>> # import as a single chromosome
>>> pop = importPopulation(format='ms', filename='ms_subPop.txt', mergeBy='subPop')

now exiting runScriptInteractively...
```

Download importMS.py

If the file format you are interested in is not supported, you can export data in csv format and convert the file by yourself. You can also try to write your own import or export functions as described in the advanced topics section of this guide.

7.2.6 Export simuPOP population in csv format (function `saveCSV`, deprecated)

Function `saveCSV` is provided in the `simuPOP.utils` module to save (the present generation of) a `simuPOP` population in comma separated formats. It allows you to save individual information fields, sex, affection status and genotype (in that order). Because this function allows you to output these information in different formats using parameters `infoFormatter`, `sexFormatter`, `affectionFormatter`, and `genoFormatter`, this function can already be used to export a `simuPOP` population to formats that are recognizable by some populat software applications. Example `saveCSV` creates a small population and demonstrates how to save it in different formats.

Example: Using function `saveCSV` to save a `simuPOP` population in different formats

```

>>> import simuPOP as sim
>>> from simuPOP.utils import saveCSV
>>> pop = sim.Population(size=[10], loci=[2, 3],
...     lociNames=['r11', 'r12', 'r21', 'r22', 'r23'],
...     alleleNames=['A', 'B'], infoFields='age')
>>> sim.initSex(pop)
>>> sim.initInfo(pop, [2, 3, 4], infoFields='age')
>>> sim.initGenotype(pop, freq=[0.4, 0.6])
>>> sim.maPenetrance(pop, loci=0, penetrance=(0.2, 0.2, 0.4))
>>> # no filename so output to standard output
>>> saveCSV(pop, infoFields='age')
age, sex, aff, r11_1, r11_2, r12_1, r12_2, r21_1, r21_2, r22_1, r22_2, r23_1, r23_2
2.0, F, A, B, B, B, B, A, B, B, A, A, B, B, A
3.0, F, U, B, A, B, A, B, A, A, A, A, B
4.0, M, U, B, B, B, B, B, B, B, B, B, B, A
2.0, M, U, B, A, B, A, B, B, B, B, B, B, A
3.0, M, A, B, B, B, B, B, B, A, A, B, A
4.0, M, U, A, B, B, A, B, B, B, B, B, B
2.0, M, U, B, B, B, B, B, B, B, B, A, A
3.0, F, U, B, B, A, A, B, B, A, A, B, B
```

(continues on next page)

(continued from previous page)

```

4.0, F, U, A, B, B, B, B, B, B, A, B, B
2.0, F, A, B, A, A, B, A, A, B, B, B, A
>>> # change affection code and how to output genotype
>>> saveCSV(pop, infoFields='age', affectionFormatter={True: 1, False: 2},
...        genoFormatter={(0,0):'AA', (0,1):'AB', (1,0):'AB', (1,1):'BB'})
age, sex, aff, r11, r12, r21, r22, r23
2.0, F, 1, BB, BB, AB, BB, AB
3.0, F, 2, AB, AB, AB, AA, AB
4.0, M, 2, BB, BB, BB, BB, AB
2.0, M, 2, AB, AB, BB, BB, AB
3.0, M, 1, BB, BB, BB, AA, AB
4.0, M, 2, AB, AB, BB, BB, BB
2.0, M, 2, BB, BB, BB, BB, AA
3.0, F, 2, BB, AA, BB, AA, BB
4.0, F, 2, AB, BB, BB, AB, BB
2.0, F, 1, AB, AB, AA, BB, AB
>>> # save to a file
>>> saveCSV(pop, filename='pop.csv', infoFields='age', affectionFormatter={True: 1,
↳False: 2},
...        genoFormatter=lambda geno: (geno[0] + 1, geno[1] + 1), sep=' ')
>>> print(open('pop.csv').read())
age sex aff r11_1 r11_2 r12_1 r12_2 r21_1 r21_2 r22_1 r22_2 r23_1 r23_2
2.0 F 1 2 2 2 2 2 2 1 2 2 2 1
3.0 F 2 2 1 2 1 2 1 1 1 1 2
4.0 M 2 2 2 2 2 2 2 2 2 2 1
2.0 M 2 2 1 2 1 2 2 2 2 2 1
3.0 M 1 2 2 2 2 2 2 1 1 2 1
4.0 M 2 1 2 2 1 2 2 2 2 2 2
2.0 M 2 2 2 2 2 2 2 2 2 1 1
3.0 F 2 2 2 1 1 2 2 1 1 2 2
4.0 F 2 1 2 2 2 2 2 2 1 2 2
2.0 F 1 2 1 1 2 1 1 2 2 2 1

now exiting runScriptInteractively...

```

[Download saveCSV.py](#)

This function is now deprecated with the introduction of function `***export**` and operator `“Exporter“`.

7.3 Module `simuPOP.demography`

7.3.1 Predefined migration models

The following functions are defined to generate migration matrixes for popular migration models.

- `migrIslandRates(r, n)` returns a migration matrix for a traditional **island model** where individuals have equal probability of migrating to any other subpopulations. This model is also called a **migrant- pool island model**.
- `migrHierarchicalIslandRates(r1, r2, n)` models a **hierarchical island model** in which local populations are grouped into neighborhoods within which there is considerable gene flow and between which there is less gene flow. should be a list of group size. is the within-group migration rate and is the cross-group migration rate. That is to say, an individual in an island has probability to stay, to be a migrant to other islands in the group (migration rate depending on the size of group), and to be a migrant to other islands in another

group (migration rate depending on the number of islands in other groups). Both can vary across groups of islands. For example, `migrHierarchicalIslandRates([r11, r12], r2, [3, 2])` returns a migration matrix

- `migrSteppingStoneRates(r, n, circular=False)` returns a migration matrix and if `circular=True`, returns
- `migr2DSteppingStoneRates(r, m, n, diagonal=False, circular=False)` models a 2D stepping stone model in which local populations are arranged into a lattice of (rows, columns) patches. The population thus needs to have subpopulations with subpopulation indexes counted by row. In this model, an individual in a center patch has a probability of to stay, and to migrate to its neighbor patches if `diagonal` is set to `False`, or to migrate to 8 neighbors (including diagonal ones) if `range` is set to 8. If `circular` is set to `False`, the corner patch has a probability of or (if `range=8`) to migrate, and a side patch has a probability or to migrate. If `circular` is set to `True`, the lattice will be conceptually connected to a ball so that there is no boundary effect. For example, for a 3 by 2 lattice

with `diagonal=False` and `circular=False`, the migration matrix will be

Many more migration models have been proposed and studied, sometimes under different names with slightly different definitions. If you cannot find your model there, it should not be too difficult to construct a migration rate matrix for it. I will be glad to add such functions to this module if you could provide a reference and your implementation of the model.

7.3.2 Uniform interface of demographic models

A realistic demographic models can be very complex that involves population growth, population bottleneck, subdivided populations, migration, population split and admixture for a typical demographic model for human populations, and carrying capacity, fecundity, sex distribution and many more factors for more complex ones (e.g. models for animal populations under continuous habitat). The goal of this module is to provide a common interface for demographic models, classes for frequently used demographic models, and several pre-defined demographic models for human populations. More complex demographic models will be added if needed.

A demographic model usually consists of the following components:

- An initial population size that is used to initialize a population (the `size` parameter of `sim.Population`)
- One or more operators to split and merge populations (e.g. Operators `SplitSubPops`)
- One or more operators to migrate individuals across subpopulations (e.g. operator `Migrator`)
- Determine sizes of subpopulations before mating (parameter `subPopSize` of a mating scheme)
- Number of generations to evolve (parameter `gen` of the `evolve` function) or operators to terminate the evolution conditionally (e.g. operator `TerminateIf`)

Using an object-oriented approach, a demographic model defined in this module encapsulates all these in a single object. More specifically, a demographic object `model` is a callable Python object that

- has attribute `model.init_size` and `model.info_fields` to determine the initial population size and required information fields to construct an initial population (e.g., `sim.Population(size=model.init_size, infoFields=model.info_fields + ['my_fields'])`)
- handles population split, merge, migration etc internally before mating when it is passed to parameter `subPopSize` of a mating scheme. (e.g. `RandomMating(subPopSize=model)`)
- has attribute `model.num_gens` to determine the number of generations to evolve (e.g. `pop.evolve(..., gen=model.num_gens)`). The model can optionally terminate the evolution by returning an empty offspring population size before mating.

- provides a function `model.plot(filename='', title='')` to plot the demographic function. It by default prints out population sizes whenever population size changes. If a `filename` is specified and if module `matplotlib` is available, it will plot the demographic model and save it to `filename`. A `title` can be specified for the figure. This function actually use the demographic model to evolve a haploid population using *RandomSelection* mating scheme, which is a good way to test if your demographic model works properly.
- saves population sizes of evolved generations, which makes it possible to revert an evolutionary process to an previous state using operator `RevertIf`.

A demographic model can be defined in two ways. The first approach is to specify the size of subpopulations at each generation, and the second approach is to specify the events that change population sizes. The *simuPOP.demography* module provides functions and classes to define demographic models using both approaches and you can use the one that is most convenient for your model.

7.3.3 Demographic models defined by outcomes

The *simuPOP.demography* module defines a number of widely used demographic models, including linear and exponential population growth with carrying capacity, shrink, split and merge, and bottleneck.

For example,

- `InstantChangeModel(T=1000, N0=1000, G=500, NG=2000)`

defines an instant population growth model that expands a population of size from 1000 to 2000 instantly at generation 500

- `InstantChangeModel(T=1000, N0=1000, G=[500, 600], NG=[100, 1000])`

defines a bottleneck model that introduces a bottleneck of size 100 between generation 500 and 600 to a population of size 1000

- `InstantChangeModel(T=1000, N0=1000, G=500, NG=[[400, 600]])`

defines a bottleneck model that split a population of size into two subpopulations of sizes 400 and 600 at generation 500

- `ExponentialGrowthModel(T=100, N0=1000, NT=10000)`

expands a population of size 1000 to 10000 in 100 generations

- `ExponentialGrowthModel(T=100, N0=[200, 800], r=[0.02, 0.01], ops=Migrator(rate=[[0, 0.1], [0.1, 0]]))`

expands a population of two subpopulation sizes at rate 0.02 and 0.01 for 100 generations, with migration between these two subpopulations. The initial population will be resized (split if necessary) to two populations of sizes 200 and 800.

- `LinearGrowthModel(N0=(200, 'A'), r=0.02, NT=1000)`

expands a population of size 200 at a rate of 0.02 (add 4 individuals at each generation) until it reaches size 1000. Here the initial size is expressed as a size name tuple, which directs the demographic model to assign the name A to the initial population. Such named size is acceptable for all places where population size is needed.

Here we specify only two of the three parameters for linear and exponential growth models and allow *simuPOP* to figure out the rest. If all three parameters are specified, the ending population size will be interpreted as carrying

capacity, namely population growth (or decline if negative rates are specified) will stop after it reaches the specified size.

A demographic model does not have to have a fixed initial population size. If an initial population size is not provided, its size will be determined from the population when it is first applied to. For example

- `InstantChangeModel(T=100, G=50, NT=[0.5, 0.5])`

splits a population into two equally sized subpopulations at generation 50. The ending population size is set to `[0.5, 0.5]`, which means 50% of the size at time G.

- `InstantChangeModel(T=100, G=50, NT=[None, 100])`

forks a population of size 100 from the main population at generation 50. `NT=[None, 100]` is equivalent to `NT=[1.0, 100]` in this case.

- `InstantChangeModel(T=0, removEmptySubPops=True)`

removes all empty subpopulations from the existing subpopulation. Here we do not specify an input population size because the size of the input population will be kept.

- `InstantChangeModel(T=0, NO=[None, 0, None], removEmptySubPops=True)`

removes the second of the three subpopulations while keep other two subpopulations intact. The input population of this demographic model must have three subpopulations.

- `ExponentialGrowthModel(T=100, NT=[10000, 20000])`

expands a population of two subpopulations to sizes 10000 and 20000 in 100 generations. An error will be raised if the population does not have two subpopulations.

- `ExponentialGrowthModel(T=100, NO=[1., 400], NT=[10000, 20000], ops=Migrator(rate=[[0, 0.1], [0.1, 0]]))`

splits a population into two subpopulations. The first one keeps all individuals (100%), the second one with 400 individuals, and then expands them, with migration, to sizes 10000 and 20000 in 100 generations.

The demography model also defines two models for population admixture. The HI model (Hybrid Isolation) model creates a separate subpopulation with individuals from two specified subpopulations. The CGF (Continuous Gene Flow) model replaces individuals from the donor population at each generation, thus keep both the recipient and donor population constant in size. For example,

- `AdmixtureModel(model=('HI', 1, 3, 0.5, 'Admixed'), T=10)`

Creates a separate population with 50% of individuals from subpopulation 1 and 50% of individuals from subpopulation 3, regardless if population sizes 1 and 3 have the same number of individuals. An optional name `Admixed` is assigned to the new subpopulation. The admixed population will evolve independently for 10 generations.

- `AdmixtureModel(model=('CGF', 1, 3, 0.9), T=10)`

Replaces 10% of individuals in subpopulation 1 with individuals from subpopulation 3 for 10 generations.

As you can imagine, these models do not provide a valid `init_size` to initialize a population. As a matter of fact, they are mostly stacked to other demographic models to form more complex demographic models, in model `MultiStageModel`. For example,


```

• MultiStageModel([
    InstantChangeModel(T=1000, N0=1000, G=[500, 600], NG=[100, 1000]),
    ExponentialGrowthModel(T=100, NT=10000)
])

```

defines a demographic model with a bottleneck followed by exponential population growth. N0 of the second stage is not specified because it is determined from its previous stage.

```

• MultiStageModel([
    LinearGrowthModel(T=100, N0=1000, r=0.01),
    ExponentialGrowthModel(T=100, N0=[0.4, 0.6], r=0.001),
    ExponentialGrowthModel(r=0.01, NT=[2000, 4000]),
    AdmixtureModel(model='HI', 0, 1, 0.8, 'admixed'), T=10)
])

```

defines a demographic model that expands a single population linearly for 100 generations, split into two subpopulations and grow exponentially at a rate of 0.001, and growth at a higher rate of 0.01 until they reaches sizes 2000 and 4000 respectively. This stage is tricky because one of the subpopulations will reach its carrying capacity sooner and keep a constant population size afterwards. As the last step, the two populations admixed and formed a new subpopulation called admixed. The model is depicted in figure [fig_multi_stage](#)

Figure: A linear and two stage exponential population growth model, followed by population admixture

Example [demoModel](#) defines a demographic model use it to evolve a population. The demographic model is depicted in Figure [fig_demoModel_example](#).

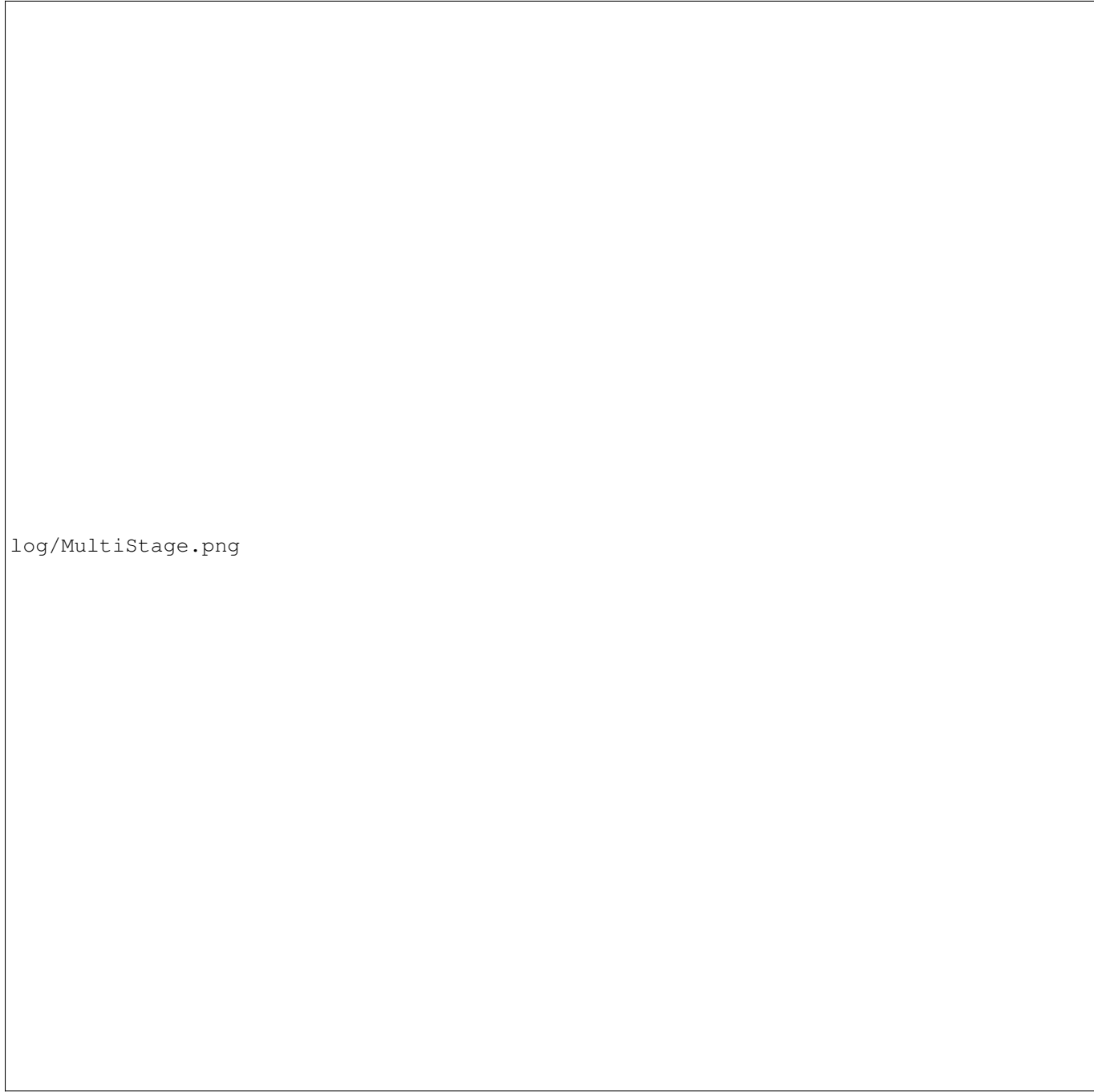
Example: A demographic model for human population

```

>>> import simuPOP as sim
>>> from simuPOP.demography import *
>>> model = MultiStageModel([
...     InstantChangeModel(T=200,
...         # start with an ancestral population of size 1000
...         N0=(1000, 'Ancestral'),
...         # change population size at 50 and 60
...         G=[50, 60],
...         # change to population size 200 and back to 1000
...         NG=[(200, 'bottleneck'), (1000, 'Post-Bottleneck')]),
...     ExponentialGrowthModel(
...         T=50,
...         # split the population into two subpopulations
...         N0=[(400, 'P1'), (600, 'P2')],
...         # expand to size 4000 and 5000 respectively
...         NT=[4000, 5000])
... ])
>>> #
>>> # model.init_size returns the initial population size
>>> # migrate_to is required for migration
>>> pop = sim.Population(size=model.init_size, loci=1,
...     infoFields=model.info_fields)
>>> pop.evolve(
...     initOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.5, 0.5])
...     ],
...     matingScheme=sim.RandomMating(subPopSize=model),
...     finalOps=

```

(continues on next page)



log/MultiStage.png

(continued from previous page)

```

...     sim.Stat(alleleFreq=0, vars=['alleleFreq_sp']),
...     gen=model.num_gens
... )
250
>>> # print out population size and frequency
>>> for idx, name in enumerate(pop.subPopNames()):
...     print('%s (%d): %.4f' % (name, pop.subPopSize(name),
...         pop.dvars(idx).alleleFreq[0][0]))
...
P1 (4000): 0.6185
P2 (5000): 0.7218
>>> # get a visual presentation of the demographic model
>>> model.plot('log/demoModel.png',
...     title='A bottleneck + exponential growth demographic model')
A bottleneck + exponential growth demographic model
0: 1000 (Ancestral)
50: 200 (bottleneck)
60: 1000 (Post-Bottleneck)
200: 419 (P1), 626 (P2)
201: 439 (P1), 653 (P2)
202: 459 (P1), 681 (P2)
203: 481 (P1), 711 (P2)
204: 504 (P1), 742 (P2)
205: 527 (P1), 774 (P2)
206: 552 (P1), 807 (P2)
207: 578 (P1), 842 (P2)
208: 605 (P1), 879 (P2)
209: 634 (P1), 917 (P2)
210: 664 (P1), 957 (P2)
211: 695 (P1), 998 (P2)
212: 728 (P1), 1041 (P2)
213: 762 (P1), 1086 (P2)
214: 798 (P1), 1133 (P2)
215: 836 (P1), 1183 (P2)
216: 875 (P1), 1234 (P2)
217: 916 (P1), 1287 (P2)
218: 960 (P1), 1343 (P2)
219: 1005 (P1), 1401 (P2)
220: 1052 (P1), 1462 (P2)
221: 1102 (P1), 1525 (P2)
222: 1154 (P1), 1591 (P2)
223: 1208 (P1), 1660 (P2)
224: 1265 (P1), 1732 (P2)
225: 1325 (P1), 1807 (P2)
226: 1387 (P1), 1885 (P2)
227: 1452 (P1), 1967 (P2)
228: 1521 (P1), 2052 (P2)
229: 1592 (P1), 2141 (P2)
230: 1667 (P1), 2234 (P2)
231: 1746 (P1), 2331 (P2)
232: 1828 (P1), 2432 (P2)
233: 1915 (P1), 2537 (P2)
234: 2005 (P1), 2647 (P2)
235: 2099 (P1), 2761 (P2)
236: 2198 (P1), 2881 (P2)
237: 2302 (P1), 3006 (P2)
238: 2410 (P1), 3136 (P2)

```

(continues on next page)

(continued from previous page)

```

239: 2524 (P1), 3272 (P2)
240: 2643 (P1), 3414 (P2)
241: 2767 (P1), 3562 (P2)
242: 2898 (P1), 3716 (P2)
243: 3034 (P1), 3877 (P2)
244: 3177 (P1), 4045 (P2)
245: 3327 (P1), 4220 (P2)
246: 3484 (P1), 4403 (P2)
247: 3648 (P1), 4593 (P2)
248: 3820 (P1), 4792 (P2)
249: 4000 (P1), 5000 (P2)
Traceback (most recent call last):
  File "/var/folders/ys/gnzk0qbx5wbdgm531v82xxljv5yqy8/T/tmpdvg5jvxd", line 2, in
↳<module>
    #begin_ignore
  File "/Users/bpeng1/anaconda3/envs/sos/lib/python3.6/site-packages/simuPOP/
↳demography.py", line 446, in plot
    region = region.reshape(region.size / 4, 4)
TypeError: 'float' object cannot be interpreted as an integer

now exiting runScriptInteractively...

```

[Download demoModel.py](#)

Figure: *A exponential population growth followed by bottleneck demographic model*

7.3.4 Demographic models defined by population changes (events)

Another way to define a demographic model is to specify the events that changes population sizes. This approach can be easier to use because it conforms with the way many demographic models are specified, also because the events can be specified for a subset of subpopulations so you can, for example, split one subpopulation without worrying about its impact on other subpopulations.

A event-based demographic model is defined using

```
EventBasedModel(events=[], T=None, N0=None, ops=[], infoFields=[])
```

where `T` and `N0` are the duration and initial size of the demographic model, respectively, and `ops` is the operators that will be applied to the population (without checking applicability). Parameter `events` accepts one or more of `DemographicEvent` and its derived classes. For example,

```
ExpansionEvent(rates=0.05, begin=500)
```

expands all subpopulations exponentially at a rate of 0.05, and

```
ExpansionEvent(rates=[0.05, 0.01], capacity=10000, subPops=[0, 2], begin=500)
```

expands two subpopulations at rates 0.05 and 0.01 respectively, until they reach 10000 individuals in each subpopulation.

```
ExpansionEvent(slopes=500, subPops=[0, 2], begin=500)
```

expands the populations linearly by adding 500 individuals to each subpopulation at each generation. These events happen at each generation starting from generation 500.



log/demoModel.png

Similarly, you can split, merge, and resize subpopulations using events `SplitEvent`, `MergeEvent`, and `ResizeEvent`. For example,

```
SplitEvent(subPops='AF', sizes=[500, 500], names=['AF', 'EU'], at=-4000)
```

splits an ancestral population named AF to two populations AF and EU at 4000 generations before the end of the demographic model. The AF population will be expanded automatically if it does not have 1000 individuals.

Finally, an `AdmixtureEvent` mix two or more subpopulations by certain proportions, and either create a new subpopulation or replace an existing subpopulation. In particular,

```
AdmixtureEvent(subPops=['MX', 'EU'], at=-10, sizes=[0.4, 0.6], name='MXL')
```

creates a new admixed population called MXL with 40% of individuals from the MX population, and the rest from the EU population. The admixture process happens once and follows an Hybrid Isolation model. Alternatively,

```
AdmixtureEvent(subPops=['MX', 'EU'], begin=-10, sizes=[0.8, 0.2], toSubPop='MX')
```

will create an admixed population with 80% MX and 20% EU individuals for 10 generations. Because 20% of the admixed population will be replaced by individuals from the EU population, this models a continuous gene flow model of admixture. If you would like to control the exact size of the admixed population, you can specify the number of individuals as integer numbers instead of proportions:

```
AdmixtureEvent(subPops=['MX', 'EU'], begin=-10, sizes=[int(1400*0.8), int(1400*0.2)],  
↳toSubPop='MX')
```

Note that the type of elements in parameter `sizes` is important, `1.` stands for all subpopulation and `1` stands for one individual from it.

Example *demoEventModel* defines the same model as *demoModel* using an event based demographic model. The result is depicted in Figure *fig_demoEventModel_example*. These two models look similar but the event-based model does not have the same final population sizes as the previous model. This is because the population size of the previous model was calculated by whereas the event based model was calculated using for each generation, and the integer rounding error accumulates over time.

Example: *A event-based demographic model*

```
>>> import simuPOP as sim
>>> from simuPOP.demography import *
>>> import math
>>> model = EventBasedModel(
...     N0=(1000, 'Ancestral'),
...     T=250,
...     events=[
...         ResizeEvent(at=50, sizes=200),
...         ResizeEvent(at=60, sizes=1000),
...         SplitEvent(sizes=[0.4, 0.6], names=['P1', 'P2'], at=200),
...         ExpansionEvent(rates=[math.log(4000/400)/50, math.log(5000/600)/50],  
↳begin=200)
...     ]
... )
>>> #
>>> # model.init_size returns the initial population size
>>> # migrate_to is required for migration
>>> pop = sim.Population(size=model.init_size, loci=1,
...     infoFields=model.info_fields)
>>> pop.evolve(
...     initOps=[
```

(continues on next page)

(continued from previous page)

```

...     sim.InitSex(),
...     sim.InitGenotype(freq=[0.5, 0.5])
... ],
... matingScheme=sim.RandomMating(subPopSize=model),
... finalOps=
...     sim.Stat(alleleFreq=0, vars=['alleleFreq_sp']),
...     gen=model.num_gens
... )
250
>>> # print out population size and frequency
>>> for idx, name in enumerate(pop.subPopNames()):
...     print('%s (%d): %.4f' % (name, pop.subPopSize(name),
...         pop.dvars(idx).alleleFreq[0][0]))
...
P1 (4000): 0.6185
P2 (5000): 0.7218
>>> # get a visual presentation of the demographic model
>>> model.plot('log/demoEventModel.png',
...     title='A event-based bottleneck + exponential growth demographic model')
A event-based bottleneck + exponential growth demographic model
0: 1000 (Ancestral)
50: 200 (Ancestral)
60: 1000 (Ancestral)
200: 419 (P1), 626 (P2)
201: 439 (P1), 653 (P2)
202: 459 (P1), 681 (P2)
203: 481 (P1), 711 (P2)
204: 504 (P1), 742 (P2)
205: 527 (P1), 774 (P2)
206: 552 (P1), 807 (P2)
207: 578 (P1), 842 (P2)
208: 605 (P1), 879 (P2)
209: 634 (P1), 917 (P2)
210: 664 (P1), 957 (P2)
211: 695 (P1), 998 (P2)
212: 728 (P1), 1041 (P2)
213: 762 (P1), 1086 (P2)
214: 798 (P1), 1133 (P2)
215: 836 (P1), 1183 (P2)
216: 875 (P1), 1234 (P2)
217: 916 (P1), 1287 (P2)
218: 960 (P1), 1343 (P2)
219: 1005 (P1), 1401 (P2)
220: 1052 (P1), 1462 (P2)
221: 1102 (P1), 1525 (P2)
222: 1154 (P1), 1591 (P2)
223: 1208 (P1), 1660 (P2)
224: 1265 (P1), 1732 (P2)
225: 1325 (P1), 1807 (P2)
226: 1387 (P1), 1885 (P2)
227: 1452 (P1), 1967 (P2)
228: 1521 (P1), 2052 (P2)
229: 1592 (P1), 2141 (P2)
230: 1667 (P1), 2234 (P2)
231: 1746 (P1), 2331 (P2)
232: 1828 (P1), 2432 (P2)
233: 1915 (P1), 2537 (P2)

```

(continues on next page)

(continued from previous page)

```

234: 2005 (P1), 2647 (P2)
235: 2099 (P1), 2761 (P2)
236: 2198 (P1), 2881 (P2)
237: 2302 (P1), 3006 (P2)
238: 2410 (P1), 3136 (P2)
239: 2524 (P1), 3272 (P2)
240: 2643 (P1), 3414 (P2)
241: 2767 (P1), 3562 (P2)
242: 2898 (P1), 3716 (P2)
243: 3034 (P1), 3877 (P2)
244: 3177 (P1), 4045 (P2)
245: 3327 (P1), 4220 (P2)
246: 3484 (P1), 4403 (P2)
247: 3648 (P1), 4593 (P2)
248: 3820 (P1), 4792 (P2)
249: 4000 (P1), 5000 (P2)
>>>

now exiting runScriptInteractively...

```

[Download demoEventModel.py](#)

Figure: *A event-based demographic model*

7.3.5 Predefined demographic models for human populations

The `simuPOP.demography` module currently defines the following models

- Out of Africa model for YRI, CEU and CHB populations (*fig_Out_of_Africa*),

```
OutOfAfricaModel(10000).plot('OutOfAfrica.png')
```

Figure: *Out of Africa model for YRI, CEU, and CHB populations*

- The settlement of new world model for Mexican American (*fig_Settlement_of_New*) (Gutenkunst, 2009, PLoS Genetics). In this model, the simulated CHB and MX populations are mixed to produce an admixed population at the last generation.

```
SettlementOfNewWorldModel(10000).plot('SettlementOfNewWorld.png')
```

Figure: *Settlement of New World model for Mexican America population*

- The demographic model developed by cosi (Schaffner, 2005, genome research).

```
CosiModel(20000).plot('Cosi.png')
```

Figure: *Demographic models for African, Asian and European populations (cosi)*

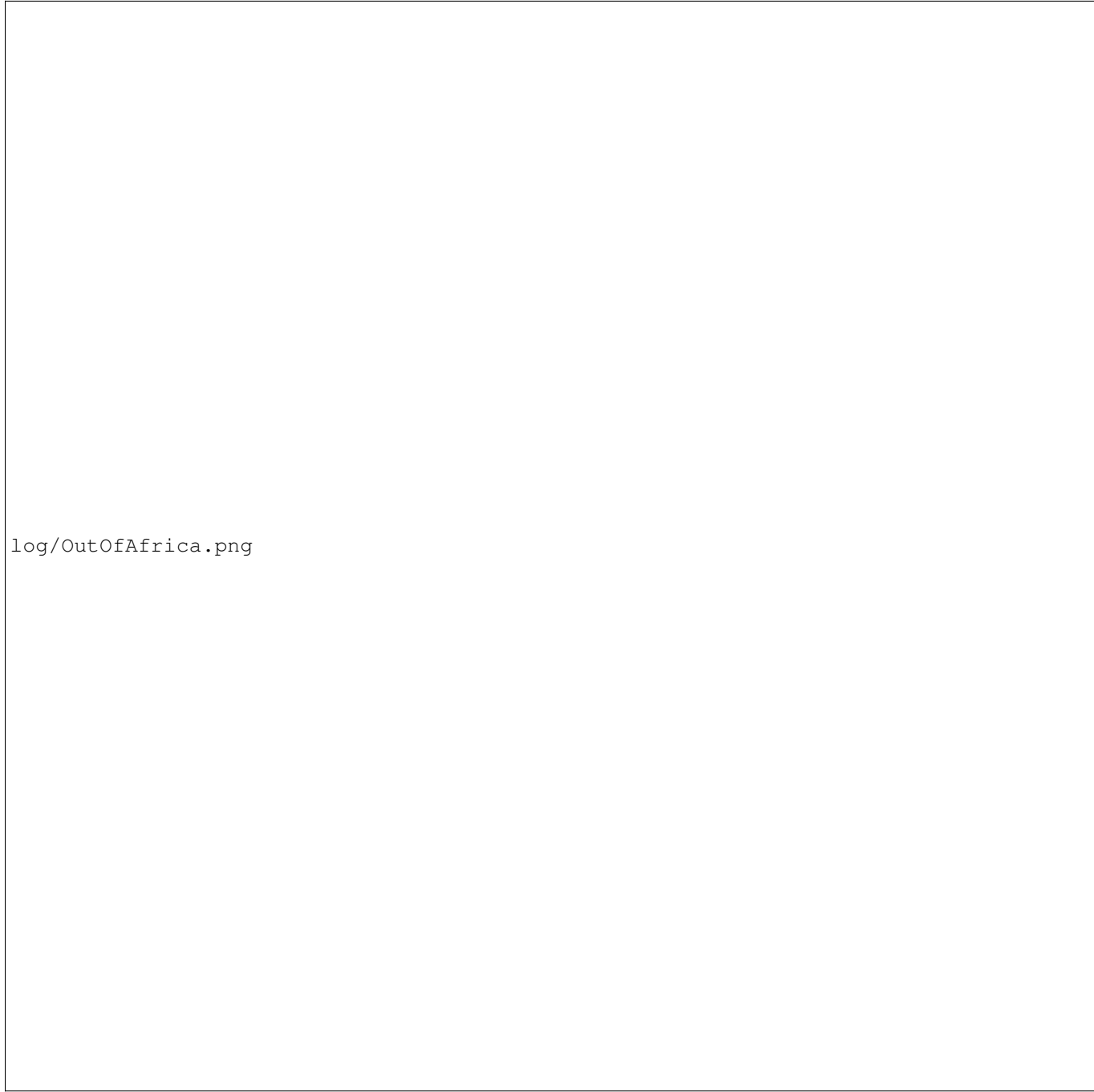
These functions all accept a parameter `scale`. If specified, it will scale all population sizes and generation numbers by the specified scaling factor. For example

```
CosiModel(20000, scale=10)
```

will result in a demographic model that evolves 2000 instead of 20000 generations, with all population sizes reduced by a factor of 10. Note that the burn-in period of the examples above are relatively short and you might need to use a longer burn-in period (e.g. T=100,000 generations for a burn-in period of about 80,000 generations).



Users/bpeng1/simuPOP/simuPOP/doc/log/demoEventModel.png



log/OutOfAfrica.png



log/SettlementOfNewWorld.png



log/Cosi.png

7.3.6 Demographic model without predefined generations to evolve *

All migration models accept one or more operators that will be applied to the population before population changes are applied. The most frequently application of this operator is to pass a migrator to the model, but we can also pass an operator to terminate a demographic model under certain conditions. For example, Example *demoTerminate* defines a demographic model that starts with a burn-in stage with indefinite size and will stop if the average allele frequency at segregating sites exceeds 0.1. It splits to two equally sized subpopulations and expand rate a rate of 0.01 to size 2000 and 5000 respectively.

Example: *A demographic model with a terminator*

```
>>> import simuPOP as sim
simuPOP Version 1.1.9 : Copyright (c) 2004-2016 Bo Peng
Revision 4583 (Oct 10 2018) for Python 3.6.6 (64bit, 0thread)
Random Number Generator is set to mt19937 with random seed 0x81aae4a664e115de.
This is the standard short allele version with 256 maximum allelic states.
For more information, please visit http://simupop.sourceforge.net,
or email simupop-list@lists.sourceforge.net (subscription required).
>>> import simuPOP.demography as demo
>>>
>>> model = demo.MultiStageModel([
...     demo.InstantChangeModel(N0=1000,
...         ops=[
...             sim.Stat(alleleFreq=sim.ALL_AVAIL, numOfSegSites=sim.ALL_AVAIL),
...             # terminate if the average allele frequency of segregating sites
...             # are more than 0.1
...             sim.TerminateIf('sum([x[1] for x in alleleFreq.values() if '
...                 'x[1] != 0])/(1 if numOfSegSites==0 else numOfSegSites) > 0.1')
...         ]
...     ),
...     demo.ExponentialGrowthModel(N0=[0.5, 0.5], r=0.01, NT=[2000, 5000])
... ])
>>>
>>> pop = sim.Population(size=model.init_size, loci=100)
>>> pop.evolve(
...     initOps=sim.InitSex(),
...     preOps=sim.SNPMutator(u=0.001, v=0.001),
...     matingScheme=sim.RandomMating(subPopSize=model),
...     postOps=[
...         sim.Stat(alleleFreq=sim.ALL_AVAIL, numOfSegSites=sim.ALL_AVAIL,
...             popSize=True, step=50),
...         sim.PyEval(r'"%d: %s, %.3f\n" % (gen, subPopSize, sum([x[1] for x '
...             'in alleleFreq.values() if x[1] != 0])/(1 if numOfSegSites == 0 '
...             'else numOfSegSites))', step=50)
...     ],
... )
0: [1000], 0.001
50: [1000], 0.047
100: [1000], 0.089
150: [738, 738], 0.128
200: [1218, 1218], 0.166
250: [2000, 2007], 0.199
300: [2000, 3310], 0.230
343
>>>

now exiting runScriptInteractively...
```

[Download demoTerminate.py](#)

7.4 Module `simuPOP.sampling`

7.4.1 Introduction

Sampling, in simuPOP term, is the action of extracting individuals from a large, potentially multi-generational, population according to certain criteria. the `simuPOP.sampling` module provides several classes and functions and allows you to define more complicated sampling schemes by deriving from its these class. For example, you can use `drawRandomSample(pop, size=100)` to select 100 random individuals from a population, or use `drawAffectedSibpairSample(pop, families=100)` to select 100 pairs of affected individuals with their parents from a multi-generational population, or a age- structured population with parents and offspring in the same generation.

The `simuPOP.sampling` module currently support random, case control, affected sibpair, nuclear family and three-generation family sampling types, and a combined sampling type that allows you to draw different types of samples. For each sampling type X, a sampler class and two functions `DrawXSample` and `DrawXSamples` are provided. The first function returns a population with all sampled individuals and the second function returns a list of sample populations.

If you would like to define your own sampling type, you can derive your sampler from one of the existing sampler classes. These sampler classes provide member functions `prepareSample`, `drawSample` and `drawSamples` and you typically only need to extend `prepareSample` of an appropriate base class.

7.4.2 Sampling individuals randomly (class `RandomSampler`, functions `drawRandomSample` and `drawRandomSamples`)

Functions `drawRandomSample` and `drawRandomSamples` draw random individuals from a given population. If a simple number is given to parameter `size`, population structure will be ignored so individuals will be drawn from all subpopulations. If a list of numbers are given, this function will draw specified numbers of individuals from each subpopulation. This function does not need parental information. If your population does not have an ID field, you will not be able to locate extracted individuals in the original population.

Example `randomSample` demonstrates how to draw a random sample from the whole population, and from each subpopulation. Because sample populations keep the population structure of the source population (this might change when parameter `subPops` is used, see a later section for details), we can use `sample.subPopSizes()` to check how many individuals are sampled from each subpopulation.

Example: *Draw random samples from a structured population*

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawRandomSample
>>> pop = sim.Population([2000]*5, loci=1)
>>> # sample from the whole population
>>> sample = drawRandomSample(pop, sizes=500)
>>> print(sample.subPopSizes())
(104, 105, 110, 81, 100)
>>> # sample from each subpopulation
>>> sample = drawRandomSample(pop, sizes=[100]*5)
>>> print(sample.subPopSizes())
(100, 100, 100, 100, 100)

now exiting runScriptInteractively...
```

[Download randomSample.py](#)

7.4.3 Sampling cases and controls (class `CaseControlSampler`, functions `CaseControlSample` and `CaseControlSamples`)

Functions `drawCaseControlSample` and `drawCaseControlSamples` draw cases (affected individuals) and controls (unaffected individuals) from a given population. If a simple number is given to parameter `cases` and `controls`, population structure will be ignored so individuals will be drawn from all subpopulations. If a list of numbers are given, this function will draw specified number of cases and controls from each subpopulation.

Example `caseControlSample` demonstrates how to draw multiple case-control samples from a population, and perform case-control association tests using the `stat` function.

Example: *Draw case control samples from a population and perform association test*

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawCaseControlSamples
>>> pop = sim.Population([10000], loci=5)
>>> sim.initGenotype(pop, freq=[0.2, 0.8])
>>> sim.maPenetrance(pop, loci=2, penetrance=[0.11, 0.15, 0.20])
>>> # draw multiple case control sample
>>> samples = drawCaseControlSamples(pop, cases=500, controls=500, numofSamples=5)
>>> for sample in samples:
...     sim.stat(sample, association=range(5))
...     print(', '.join(['%.6f' % sample.dvars().Allele_ChiSq_p[x] for x in_
↪range(5)]))
...
0.694748, 0.333041, 0.001039, 0.078127, 0.774085
0.261750, 0.954592, 0.031830, 0.737788, 0.865679
0.954949, 0.371093, 0.092487, 0.622153, 0.075739
0.654721, 0.433848, 0.002859, 0.696375, 0.956630
0.439721, 1.000000, 0.069651, 0.471087, 0.238199

now exiting runScriptInteractively...
```

[Download caseControlSample.py](#)

7.4.4 Sampling Pedigrees (functions `indexToID` and `plotPedigree`)

If your sampling scheme involves parental information, you need to prepare your population so that it has

- an ID field (usually `'ind_id'`) that stores a unique ID for each individual.
- two information fields (usually `'father_id'`, and `'mother_id'`) that stores the ID of parents of each individual. Although simuPOP supports one- parent Pedigrees, this feature will not be discussed in this guide.

The preferred method to prepare such a population is to add information fields `ind_id`, `father_id` and `mother_id` to a population and track ID based Pedigrees during evolution. More specifically, you can use operators `IdTagger` and `PedigreeTagger` to assign IDs and record parental IDs of each offspring during mating. This method supports age-structured population when parents and offspring can be stored in the same generation.

You can also use information fields `father_idx` and `mother_idx` and operator `ParentsTagger` to track indexes of parents in the parental generations. Before sampling, you can use function `indexToID` to add needed information fields and convert index based parental relationship to ID based relationship. Because parents have to stay in ancestral generations, this method does not support age-structured population.

If you have R and rpy installed on your system, you can install the `kinship` library of R and use it to analyze Pedigree. The `simuPOP.sampling` module provides a function `plotPedigree` to use this library to plot Pedigrees. Example `plotPedigree` demonstrates how to use function `sampling.indexToID` to prepare a pedigree and how to use `sampling.DrawPedigree` to plot it.

Figure `fig_Pedigree` plots a small three-generational population with 15 individuals at each generation. It is pretty clear that random mating produces bad pedigree structure because it is common that one parent would have multiple spouses.

7.4.5 Sampling affected sibpairs (class `AffectedSibpairSampler`, functions `drawAffectedSibpairSample(s)`)

An affected sibpair family consists of two parents and their affected offspring. Such families are useful in linkage analysis because of high likelihood of shared disease predisposing alleles between siblings. `simuPOP.sampling` module provides functions `drawAffectedSibpairSample` and `drawAffectedSibpairSamples` to draw such families from a population. Example `sampleAffectedSibpair` draws two affected sibpair from the pedigree created in Example `plotPedigree`, with samples plotted in Figure `fig_affectedSibpair`.

Example: *Draw affected sibpairs from a population*

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import indexToID
>>> pop = sim.Population(size=15, loci=5, infoFields=['father_idx', 'mother_idx'],
↳ ancGen=2)
>>> pop.evolve(
...     preOps=[
...         sim.InitSex(),
...         sim.InitGenotype(freq=[0.7, 0.3]),
...     ],
...     matingScheme=sim.RandomMating(numOffspring=(sim.UNIFORM_DISTRIBUTION, 2, 4),
...     ops=[sim.MendelianGenoTransmitter(), sim.ParentsTagger()]),
...     postOps=sim.MaPenetrance(loci=3, penetrance=(0.1, 0.4, 0.7)),
...     gen = 5
... )
5
>>> indexToID(pop, reset=True)
>>> # three information fields were added
>>> print(pop.infoFields())
('father_idx', 'mother_idx', 'ind_id', 'father_id', 'mother_id')
>>> # save this population for future use
>>> pop.save('log/pedigree.pop')
>>>
>>> from simuPOP.sampling import drawAffectedSibpairSample
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawAffectedSibpairSample(pop, families=2)
Warning: number of requested Pedigrees 2 is greater than what exists (0).
Warning: not enough non-overlapping Pedigrees are found (requested 2, found 0).

now exiting runScriptInteractively...
```

[Download sampleAffectedSibpair.py](#)

7.4.6 Sampling nuclear families (class `NuclearFamilySampler`, functions `drawNuclearFamilySample` and `drawNuclearFamilySamples`)

A nuclear family consists of two parents and their offspring. Functions `drawNuclearFamilySample` and `drawNuclearFamilySamples` to draw such families from a population, with restrictions on number of offspring, number of affected parents and number of affected offspring. Although fixed numbers could be given, a range with minimal and maximal acceptable numbers are usually provided. Example *sampleNuclearFamily* draws two nuclear families from the pedigree created in Example `plotPedigree`. The samples are plotted in Figure `fig_nuclearFamily`.

Example: Draw nuclear families from a population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawNuclearFamilySample
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawNuclearFamilySample(pop, families=2, numOffspring=(2,4),
...     affectedParents=(1,2), affectedOffspring=(1, 3))
Warning: number of requested Pedigrees 2 is greater than what exists (0).
Warning: not enough non-overlapping Pedigrees are found (requested 2, found 0).
>>> # try to separate two families?
>>> sample.asPedigree()
>>> #= sim.Pedigree(sample, loci=sim.ALL_AVAIL, infoFields=sim.ALL_AVAIL)
>>> sample.addInfoFields('ped_id')
>>> # return size of families
>>> sz = sample.identifyFamilies(pedField='ped_id')
>>> print(sz)
()
>>> ped1 = sample.extractIndividuals(IDs=0, idField='ped_id')
>>> # print the ID of all individuals in the first pedigree
>>> print([ind.ind_id for ind in ped1.allIndividuals()])
[]

now exiting runScriptInteractively...
```

[Download sampleNuclearFamily.py](#)

7.4.7 Sampling three-generation families (class `ThreeGenFamilySampler`, functions `drawThreeGenFamilySample` and `drawThreeGenFamilySamples`)

A three-generation family consists of two parents, their common offspring, offspring's spouses, and their common offspring (grandchildren). individuals in sampled families have either no or two parents. Functions `drawThreeGenFamilySample` and `drawThreeGenFamilySamples` to draw such families from a population, with restrictions on number of offspring, total number of individuals and number of affected individuals in the Pedigree. These parameters (`numOffspring`, `pedSize` and `numAffected`) could be a fixed number of a range with minimal and maximal acceptable numbers. Example *sampleNuclearFamily* draws two three generation families from the pedigree created in Example `plotPedigree`. The samples are plotted in Figure `fig_nuclearFamily`.

Example: Draw three-generation families from a population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawThreeGenFamilySample
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawThreeGenFamilySample(pop, families=2, numOffspring=(1, 3),
...     pedSize=(8, 15), numOfAffected=(2, 5))

now exiting runScriptInteractively...
```

[Download sampleThreeGenFamily.py](#)

7.4.8 Sampling different types of samples (class `CombinedSampler`, functions `drawCombinedSample` and `drawCombinedSamples`)

Samples in real world studies sometimes do not have uniform types so it is useful to draw samples of different types from the same population. Although it is possible to draw samples using different functions and combine them, handling of overlapping individuals, namely individuals who are chosen by multiple samplers, can be a headache. The combined sampler of `simuPOP.sampling` is designed to overcome this problem. This sampler takes a list of sampler objects and apply them to a population sequentially. The extracted sample will not have overlapping individuals.

Example `combinedSampling` draws an affected sibpair family and a nuclear family from the pedigree created in Example `plotPedigree`. The samples are plotted in Figure `combinedSampling`.

Example: Draw different types of samples from a population

```
>>> import simuPOP as sim
>>> from simuPOP.sampling import drawCombinedSample, AffectedSibpairSampler, \
↳ NuclearFamilySampler
>>> pop = sim.loadPopulation('log/pedigree.pop')
>>> sample = drawCombinedSample(pop, samplers = [
...     AffectedSibpairSampler(families=1),
...     NuclearFamilySampler(families=1, numOffspring=(2,4), affectedParents=(1,2), \
↳ affectedOffspring=(1,3))
... ])
Warning: number of requested Pedigrees 1 is greater than what exists (0).
Warning: not enough non-overlapping Pedigrees are found (requested 1, found 0).
Warning: number of requested Pedigrees 1 is greater than what exists (0).
Warning: not enough non-overlapping Pedigrees are found (requested 1, found 0).

now exiting runScriptInteractively...
```

[Download combinedSampling.py](#)

7.4.9 Sampling from subpopulations and virtual subpopulations *

Virtual subpopulations (VSPs) could be specified in the `subPops` parameter of sampling classes and functions. This can be used to limit your samples to individuals with certain properties. For example, you may want to match the age of cases and controls in a case-control association study by selecting your samples from a certain age group. For examples, Example `samplingVSP` draws 500 cases and 500 controls from two a VSP with individual ages between 40 and 60.

Example: Draw samples from a virtual subpopulation.

```
>>> import simuPOP as sim
>>> # create an age-structured population with a disease
>>> import random
>>> pop = sim.Population(10000, loci=10, infoFields='age')
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.initInfo(pop, lambda: random.randint(0, 70), infoFields='age')
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=(40, 60), field='age'))
>>> sim.maPenetrance(pop, loci=5, penetrance=(0.1, 0.2, 0.3))
>>> #
>>> from simuPOP.sampling import drawCaseControlSample
>>> sample = drawCaseControlSample(pop, cases=500, controls=500, subPops=[(0,1)])
>>> ageInSample = sample.indInfo('age')
>>> print(min(ageInSample), max(ageInSample))
```

(continues on next page)

(continued from previous page)

```
40.0 59.0

now exiting runScriptInteractively...
```

Download `samplingVSP.py`

If a list of sample sizes is given, specified number of samples will be drawn from each subpopulation. For example, if you have an age-structured population when individuals with different ages have different risk to a disease, you might want to draw affected individuals from different age groups and perform association analyses. Function `drawCaseControlSample` cannot be used because both groups are affected, but you can `drawRandomSample` from two VSPs defined by age. Example *samplingSeparateVSPs* demonstrates how to use this method.

Example: *Sampling separately from different virtual subpopulations*

```
>>> import simuPOP as sim
>>> # create an age-structured population with a disease
>>> import random
>>> pop = sim.Population(10000, loci=10, infoFields='age')
>>> sim.initGenotype(pop, freq=[0.3, 0.7])
>>> sim.initInfo(pop, lambda: random.randint(0, 70), infoFields='age')
>>> pop.setVirtualSplitter(sim.InfoSplitter(cutoff=(20, 40), field='age'))
>>> # different age group has different penetrance
>>> sim.maPenetrance(pop, loci=5, penetrance=(0.1, 0.2, 0.3), subPops=[(0,1)])
>>> sim.maPenetrance(pop, loci=5, penetrance=(0.2, 0.4, 0.6), subPops=[(0,2)])
>>> # count the number of affected individuals in each group
>>> sim.stat(pop, numOfAffected=True, subPops=[(0,1), (0,2)], vars='numOfAffected_sp')
>>> print(pop.dvars((0,1)).numOfAffected, pop.dvars((0,2)).numOfAffected)
668 2215
>>> #
>>> from simuPOP.sampling import drawRandomSample
>>> sample = drawRandomSample(pop, sizes=[500, 500], subPops=[(0,1), (0,2)])
>>> # virtual subpopulations are rearranged to different subpopulations.
>>> print(sample.subPopSizes())
(500, 500)

now exiting runScriptInteractively...
```

Download `samplingSeparateVSPs.py`

7.5 Module `simuPOP.gsl`

simuPOP makes use of many functions from the GUN Scientific Library. These functions are used to generate random number and perform statistical tests within simuPOP. Although these functions are not part of simuPOP, they can be useful to users of simuPOP from time to time and it makes sense to expose these functions directly to users.

Module `simuPOP.gsl` contains a number of GSL functions. Because only a small proportion of GSL functions are used in simuPOP, this module is by no means a comprehensive wrapper of GSL. Please refer to the simuPOP reference manual for a list of functions included in this module, and the GSL manual for more details. Because random number generation functions such as `gsl_ran_gamma` are already provided in the `simuPOP.RNG` class (e.g. `getRNG.randGamma`), they are not provided in this module.

A real world example

Previous chapters use a lot of examples to demonstrate individual simuPOP features. However, it might not be clear how to integrate these features in longer scripts that address real world problems, which may involve larger populations, more complex genetic and demographic models and may run thousands of replicates with different parameters. This chapter will show you, step by step, how to write a complete simuPOP script that has been used in a real-world research topic.

8.1 Simulation scenario

Reich and Lander Reich2001a proposed a population genetics framework to model the evolution of allelic spectra (the number and population frequency of alleles at a locus). The model is based on the fact that human population grew quickly from around 10,000 to 6 billion in 18,000 -150,000 years. His analysis showed that at the founder population, both common and rare diseases have simple spectra. After the sudden expansion of population size, the allelic spectra of simple diseases become complex; while those of complex diseases remained simple.

This example is a simplified version of the `simuCDCV.py` script that simulates this evolution process and observe the allelic spectra of both types of diseases. The complete script is available at <http://simupop.sourceforge.net/cookbookthe> simuPOP online cookbook. The results are published in Peng2007, which has much more detailed discussion about the simulations, and the parameters used.

8.2 Demographic model

The original paper used a very simple instant population growth model. Under the model assumption, a population with an initial population size would evolve generations, instantly expand its population size to and evolve another generations. Such a model can be easily implemented as follows:

```
def ins_expansion(gen):
    'An instant population growth model'
    if gen < G0:
        return NO
```

(continues on next page)

(continued from previous page)

```

else:
    return N1

```

Other demographic models could be implemented similarly. For example, an exponential population growth model that expand the population size from N_0 to N_1 in generations could be defined as

```

def exp_expansion(gen):
    'An exponential population growth model'
    if gen < G0:
        return N0
    else:
        rate = (math.log(N1) - math.log(N0))/G1
        return int(N0 * math.exp((gen - G0) * rate))

```

That is to say, we first solve for t and then calculate for a given generation.

There is a problem here: the above definitions treat N_0 , G_0 , N_1 and G_1 as global variables. This is OK for small scripts but is certainly not a good idea for larger scripts especially when different parameters will be used. A better way is to wrap these functions by another function that accept N_0 , G_0 , N_1 and G_1 as parameters. That is demonstrated in Example *reichDemo* where a function `demo_model` is defined to return either an instant or an exponential population growth demographic function.

Example: *A demographic function producer*

```

>>> import simuPOP as sim
>>> import math
>>> def demo_model(model, N0=1000, N1=100000, G0=500, G1=500):
...     '''Return a demographic function
...     model: linear or exponential
...     N0:    Initial sim.population size.
...     N1:    Ending sim.population size.
...     G0:    Length of burn-in stage.
...     G1:    Length of sim.population expansion stage.
...     '''
...     def ins_expansion(gen):
...         if gen < G0:
...             return N0
...         else:
...             return N1
...     rate = (math.log(N1) - math.log(N0))/G1
...     def exp_expansion(gen):
...         if gen < G0:
...             return N0
...         else:
...             return int(N0 * math.exp((gen - G0) * rate))
...     if model == 'instant':
...         return ins_expansion
...     elif model == 'exponential':
...         return exp_expansion
...
>>> # when needed, create a demographic function as follows
>>> demo_func = demo_model('exponential', 1000, 100000, 500, 500)
>>> # sim.population size at generation 700
>>> print(demo_func(700))
6309

now exiting runScriptInteractively...

```

[Download reichDemo.py](#)

Note: The defined demographic functions return the total population size (a number) at each generation because no subpopulation is considered. A list of subpopulation sizes should be returned if there are more than one subpopulations.

8.3 Mutation and selection models

The theoretical model employs an infinite allele model where there is a single wild type allele and an infinite number of disease alleles. Each mutation would introduce a new disease allele and there is no back mutation (mutation from disease allele to wild type allele).

This mutation model can be mimicked by a k -allele model with reasonably large k . We initialize all alleles to 0 which is the wild type () and all other alleles are considered as disease alleles (). Because an allele in a k -allele mutation model can mutate to any other allele with equal probability, since there are many more disease alleles than the wild type allele. If we choose a smaller (e.g. 10), recurrent and back mutations can no longer be ignored but it would be interesting to simulate such cases because they are more realistic than the infinite allele model in some cases.

A k -allele model can be simulated using the `KAlleleMutator` operator which accepts a mutation rate and a maximum allelic state as parameters.

```
KAlleleMutator(k=k, rates=mu)
```

Because there are many possible disease alleles, a multi-allelic selector (`MaSelector`) could be used to select against the disease alleles. This operator accepts a single or a list of wild type alleles ([0] in this case) and treats all other alleles as disease alleles. A penetrance table is needed which specifies the fitness of each individual when they have 0, 1 or 2 disease alleles respectively. In this example, we assume a recessive model in which only genotype causes genetic disadvantages. If we assume a selection pressure parameter s , the operator to use is

```
MaSelector(loci=0, wildtype=0, penetrance=[1, 1, 1-s])
```

Note that the use of this selector requires a population information field `fitness`.

This example uses a single-locus selection model but the complete script allows the use of different kinds of multi-locus selection model. If we assume a multiplicative multi-locus selection model where fitness values at different loci are combined (multiplied), a multi-locus selection model (`MlSelector`) could be used as follows:

```
MlSelector([
    MaSelector(loci=loc1, fitness=[1,1,1-s1], wildtype=0),
    MaSelector(loci=loc2, fitness=[1,1,1-s2], wildtype=0)],
    mode=MULTIPLICATIVE
)
```

These multi-locus models treat disease alleles at different loci more or less independently. If more complex multi-locus models (e.g. models involve gene - gene and/or gene - interaction) are involved, a multi-locus selector that uses a multi-locus penetrance table could be used.

8.4 Output statistics

We first want to output total disease allele frequency of each locus. This is easy because `Stat()` operator can calculate allele frequency for us. What we need to do is use a `Stat()` operator to calculate allele frequency and get the result from population variable `alleleFreq`. Because allele frequencies add up to one, we can get the total disease allele frequency using the allele frequency of the wild type allele 0 (). The actual code would look more or less like this:

```
Stat(alleleFreq=[0,1]),
PyEval(r'("%.2f" % (1-alleleFreq[0][0]))')
```

We are also interested in the effective number of alleles Reich2001a at a locus. Because simuPOP does not provide an operator or function to calculate this statistic, we will have to calculate it manually. Fortunately, this is not difficult because effective number of alleles can be calculated from existing allele frequencies, using formula

where f is the allele frequency of disease allele.

A quick-and-dirty way to output at a locus (e.g. locus 0) can be:

```
PyEval('1./sum([(alleleFreq[0][x]/(1-alleleFreq[0][0]))**2 for x in alleleFreq[0].
↪keys() if x != 0])')
```

but this expression looks complicated and does not handle the case when $f = 1$. A more robust method would involve the `stmts` parameter of `PyEval`, which will be evaluated before parameter `expr`:

```
PyEval(stmts='''if alleleFreq[0][0] == 1:
    ne = 0
else:
    freq = [freq[0][x] for x in alleleFreq[0].keys() if x != 0]
    ne = 1./sum([(f/(1-alleleFreq[0][0]))**2 for x in freq])
''', expr=r'("%.3f" % ne)')
```

However, this piece of code does not look nice with the multi-line string, and the operator is not really reusable (only valid for locus 0). It makes sense to define a function to calculate generally:

```
def ne(pop, loci):
    ' calculate effective number of alleles at given loci'
    stat(pop, alleleFreq=loci)
    ne = {}
    for loc in loci:
        freq = [y for x,y in pop.dvars().alleleFreq[loc].iteritems() if x != 0]
        sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
        if sumFreq == 0:
            ne[loc] = 0
        else:
            ne[loc] = 1. / sum([(x/sumFreq)**2 for x in freq])
    # save the result to the population.
    pop.dvars().ne = ne
    return True
```

When it is needed to calculate effective number of alleles, a Python operator that uses this function can be used. For example, operator

```
PyOperator(func=ne, param=[0], step=5)
PyEval(r'("%.3f" % ne[0]', step=5)
```

would calculate effective number of alleles at locus 0 and output it.

The biggest difference between `PyEval` and `PyOperator` is that `PyOperator` is no longer evaluated in the population's local namespace. You will have to get the variables explicitly using the `pop.dvars()` function, and the results have to be explicitly saved to the population's local namespace.

The final implementation, as a way to demonstrate how to define a new statistics that hides all the details, defines a new operator by inheriting a class from `PyOperator`. The resulting operator could be used as a regular operator (e.g., `ne(loci=[0])`). A function `Ne` is also defined as the function form of this operator. The code is listed in Example [reichstat](#)

Example: A customized operator to calculate effective number of alleles

```
>>> import simuPOP as sim
>>> class ne(sim.PyOperator):
...     '''Define an operator that calculates effective number of
...     alleles at given loci. The result is saved in a population
...     variable ne.
...     '''
...     def __init__(self, loci, *args, **kwargs):
...         self.loci = loci
...         sim.PyOperator.__init__(self, func=self.calcNe, *args, **kwargs)
...     #
...     def calcNe(self, pop):
...         sim.stat(pop, alleleFreq=self.loci)
...         ne = {}
...         for loc in self.loci:
...             freq = pop.dvars().alleleFreq[loc]
...             sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
...             if sumFreq == 0:
...                 ne[loc] = 0
...             else:
...                 ne[loc] = 1. / sum([(freq[x]/sumFreq)**2 for x in list(freq.
→keys()) if x != 0])
...         # save the result to the sim.Population.
...         pop.dvars().ne = ne
...         return True
...
>>> def Ne(pop, loci):
...     '''Function form of operator ne'''
...     ne(loci).apply(pop)
...     return pop.dvars().ne
...
>>> pop = sim.Population(100, loci=[10])
>>> sim.initGenotype(pop, freq=[.2] * 5)
>>> print(Ne(pop, loci=[2, 4]))
{2: 3.9565470135154768, 4: 3.948841408365935}

now exiting runScriptInteractively...
```

[Download reichstat.py](#)

8.5 Initialize and evolve the population

With appropriate operators to perform mutation, selection and output statistics, it is relatively easy to write a simulator to perform a simulation. This simulator would create a population, initialize alleles with an initial allelic spectrum, and then evolve it according to specified demographic model. During the evolution, mutation and selection will be applied, statistics will be calculated and outputted.

Example: Evolve a population subject to mutation and selection

```
>>> import simuPOP as sim
>>>
>>>
>>> def simulate(model, N0, N1, G0, G1, spec, s, mu, k):
...     '''Evolve a sim.Population using given demographic model
...     and observe the evolution of its allelic spectrum.
```

(continues on next page)

(continued from previous page)

```

...     model: type of demographic model.
...     N0, N1, G0, G1: parameters of demographic model.
...     spec: initial allelic spectrum, should be a list of allele
...           frequencies for each allele.
...     s: selection pressure.
...     mu: mutation rate.
...     k: k for the k-allele model
...     '''
...     demo_func = demo_model(model, N0, N1, G0, G1)
...     pop = sim.Population(size=demo_func(0), loci=1, infoFields='fitness')
...     pop.evolve(
...         initOps=[
...             sim.InitSex(),
...             sim.InitGenotype(freq=spec, loci=0)
...         ],
...         matingScheme=sim.RandomMating(subPopSize=demo_func),
...         postOps=[
...             sim.KAlleleMutator(k=k, rates=mu),
...             sim.MaSelector(loci=0, fitness=[1, 1, 1 - s], wildtype=0),
...             ne(loci=[0], step=100),
...             sim.PyEval(r'%d: %.2f\t%.2f\n' % (gen, 1 - alleleFreq[0][0], ne[0])),
...             step=100),
...         ],
...         gen = G0 + G1
...     )
>>> simulate('instant', 1000, 10000, 500, 500, [0.9]+[0.02]*5, 0.01, 1e-4, 200)
0: 0.09      4.91
100: 0.12    2.63
200: 0.09    1.22
300: 0.02    2.85
400: 0.02    2.12
500: 0.05    1.02
600: 0.06    1.51
700: 0.08    1.58
800: 0.09    1.80
900: 0.08    1.79

now exiting runScriptInteractively...

```

[Download reichEvolve.py](#)

8.6 Option handling

Everything seems to be perfect until you need to

1. Run more simulations with different parameters such as initial population size and mutation rate. This requires the script to get its parameters from command line (or a configuration file) and executes in batch mode, perhaps on a cluster system.
2. Allow users who are not familiar with the script to run it. This would better be achieved by a graphical user interface.
3. Allow other Python scripts to import your script and run the simulation function directly.

Although a number of Python modules such as `getopt` are available, the simuPOP *simuOpt* module is especially

designed to allow a simuPOP script to be run both in batch and in GUI mode, in standard and optimized mode. Example *reich* makes use of this module.

Example: A complete simulation script

```
#!/usr/bin/env python
#
# Author: Bo Peng
# Purpose: A real world example for simuPOP user's guide.
#
'''
Simulation the evolution of allelic spectra (number and frequencies
of alleles at a locus), under the influence of sim.population expansion,
mutation, and natural selection.
'''
import simuOpt
simuOpt.setOptions(quiet=True, alleleType='long')
import simuPOP as sim
import sys, types, os, math
options = [
    {'name': 'demo',
     'default': 'instant',
     'label': 'Population growth model',
     'description': 'How does a sim.Population grow from N0 to N1.',
     'type': ('chooseOneOf', ['instant', 'exponential']),
    },
    {'name': 'N0',
     'default': 10000,
     'label': 'Initial sim.population size',
     'type': 'integer',
     'description': '''Initial sim.population size. This size will be maintained
till the end of burnin stage''',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'N1',
     'default': 100000,
     'label': 'Final sim.population size',
     'type': 'integer',
     'description': 'Ending sim.population size (after sim.population expansion)',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'G0',
     'default': 500,
     'label': 'Length of burn-in stage',
     'type': 'integer',
     'description': 'Number of generations of the burn in stage.',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'G1',
     'default': 1000,
     'label': 'Length of expansion stage',
     'type': 'integer',
     'description': 'Number of geneartions of the sim.population expansion stage',
     'validator': simuOpt.valueGT(0)
    },
    {'name': 'spec',
     'default': [0.9] + [0.02]*5,
     'label': 'Initial allelic spectrum',
```

(continues on next page)

(continued from previous page)

```

        'type': 'numbers',
        'description': '''Initial allelic spectrum, should be a list of allele
            frequencies, for allele 0, 1, 2, ... respectively.''' ,
        'validator': simuOpt.valueListOf(simuOpt.valueBetween(0, 1)),
    },
    { 'name': 's',
      'default': 0.01,
      'label': 'Selection pressure',
      'type': 'number',
      'description': '''Selection coefficient for homozygtes (aa) genotype.
          A recessive selection model is used so the fitness values of
          genotypes AA, Aa and aa are 1, 1 and 1-s respectively.''' ,
      'validator': simuOpt.valueGT(-1),
    },
    { 'name': 'mu',
      'default': 1e-4,
      'label': 'Mutation rate',
      'type': 'number',
      'description': 'Mutation rate of a k-allele mutation model',
      'validator': simuOpt.valueBetween(0, 1),
    },
    { 'name': 'k',
      'default': 200,
      'label': 'Maximum allelic state',
      'type': 'integer',
      'description': 'Maximum allelic state for a k-allele mutation model',
      'validator': simuOpt.valueGT(1),
    },
]

def demo_model(type, N0=1000, N1=100000, G0=500, G1=500):
    '''Return a demographic function
    type: linear or exponential
    N0:    Initial sim.population size.
    N1:    Ending sim.population size.
    G0:    Length of burn-in stage.
    G1:    Length of sim.population expansion stage.
    '''
    rate = (math.log(N1) - math.log(N0)) / G1
    def ins_expansion(gen):
        if gen < G0:
            return N0
        else:
            return N1

    def exp_expansion(gen):
        if gen < G0:
            return N0
        else:
            return int(N0 * math.exp((gen - G0) * rate))

    if type == 'instant':
        return ins_expansion
    elif type == 'exponential':
        return exp_expansion

class ne(sim.PyOperator):

```

(continues on next page)

(continued from previous page)

```

'''Define an operator that calculates effective number of
alleles at given loci. The result is saved in a population
variable ne.
'''
def __init__(self, loci, *args, **kwargs):
    self.loci = loci
    sim.PyOperator.__init__(self, func=self.calcNe, *args, **kwargs)

def calcNe(self, pop):
    sim.stat(pop, alleleFreq=self.loci)
    ne = {}
    for loc in self.loci:
        freq = pop.dvars().alleleFreq[loc]
        sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
        if sumFreq == 0:
            ne[loc] = 0
        else:
            ne[loc] = 1. / sum([(freq[x]/sumFreq)**2 for x in list(freq.keys())
↪if x != 0])
        # save the result to the sim.Population.
        pop.dvars().ne = ne
    return True

def simuCDCV(model, N0, N1, G0, G1, spec, s, mu, k):
    '''Evolve a sim.Population using given demographic model
    and observe the evolution of its allelic spectrum.
    model: type of demographic model.
    N0, N1, G0, G1: parameters of demographic model.
    spec: initial allelic spectrum, should be a list of allele
        frequencies for each allele.
    s: selection pressure.
    mu: mutation rate.
    k: maximum allele for the k-allele model
    '''
    demo_func = demo_model(model, N0, N1, G0, G1)
    print(demo_func(0))
    pop = sim.Population(size=demo_func(0), loci=1, infoFields='fitness')
    pop.evolve(
        initOps=[
            sim.InitSex(),
            sim.InitGenotype(freq=spec, loci=0)
        ],
        matingScheme=sim.RandomMating(subPopSize=demo_func),
        postOps=[
            sim.KAlleleMutator(rates=mu, k=k),
            sim.MaSelector(loci=0, fitness=[1, 1, 1 - s], wildtype=0),
            ne(loci=(0,), step=100),
            sim.PyEval(r'"%d: %.2f\t%.2f\n" % (gen, 1 - alleleFreq[0][0], ne[0])',
                        step=100),
        ],
        gen = G0 + G1
    )
    return pop

if __name__ == '__main__':
    # get parameters
    par = simuOpt.Params(options, __doc__)

```

(continues on next page)

(continued from previous page)

```

if not par.getParam():
    sys.exit(1)

if not sum(par.spec) == 1:
    print('Initial allelic spectrum should add up to 1.')
    sys.exit(1)
# save user input to a configuration file
par.saveConfig('simuCDCV.cfg')
#
simuCDCV(*par.asList())

```

Download simuCDCV.py

Example *reich* uses a programming style that is used by almost all simuPOP scripts. I highly recommend this style because it makes your script self-documentary and work well under a variety of environments. A script written in this style follows the following order:

1. First comment block

The first line of the script should always be

```
#!/usr/bin/env python
```

This line tells a Unix shell which program should be used to process this script if the script is set to be executable. This line is ignored under windows. It is customary to put author and date information at the top of a script as Python comments.

2. Module doc string

The first string in a script is the module docstring, which can be referred by variable `__doc__` in the script. It is a good idea to describe what this script does in detail here. As you will see later, this docstring will be used in the `simuOpt.getParam()` function and be outputted in the usage information of the script.

3. Loading simuPOP and other Python modules

simuPOP and other modules are usually imported after module docstring. This is where you specify which simuPOP module to use. Although a number of parameters could be used, usually only `alleleType` is specified because other parameters such as `gui` and `optimized` should better be controlled from command line.

4. Parameter description list

A list of parameter description dictionaries are given here. This list specifies what parameters will be used in this script and describes the type, default value, name of command line option, label of the parameter in the parameter input dialog in detail. Although some dictionary items can be ignored, it is a good practice to give detailed information about each parameter here.

5. Helper functions and classes

Helper functions and classes are given before the main simulation function.

6. Main simulation function

The main simulation function performs the main functionality of the whole script. It is written as a function so that it can be imported and executed by another script. The parameter processing part of the script would be ignored in this case.

7. Script execution part conditioned by `__name__ == '__main__'`

The execution part of a script should always be inside of a `if __name__ == '__main__'` block so that the script will not be executed when it is imported by another script. The first few lines of this execution block

are almost always

```
par = simuOpt.Params(options, __doc__)
if not par.getParam():
    sys.exit(1)
```

which creates a `simuOpt` object and tries to get parameters from command line option, a configuration file, a parameter input dialog or interactive user input, depending on how this script is executed. Optionally, you can use

```
par.saveConfig('file.cfg')
```

to save the current configuration to a file so that the same parameters could be retrieved later using parameter `--config file.cfg`.

After simply parameter validation, the main simulation function can be called. This example uses `simuCDCV(*par.asList())` because the parameter list in the `par` object match the parameter list of function `simuCDCV` exactly. If there are a large number of parameters, it may be better to pass the `simuOpt` object directly in the main simulation function.

The script written in this style could be executed in a number of ways.

1. If a user executes the script directly, a Tkinter or wxPython dialog will be displayed for users to input parameters. This parameter is shown in Figure [fig_simuCDCV_dialog](#).

Figure: *Parameter input dialog of the `simuCDCV` script*

2. The help message of this script could be displayed using the Help button of the parameter input dialog, or using command `simuCDCV.py -h`.
3. Using parameter `--gui=False`, the script will be run in batch mode. You can specify parameters using

```
simuCDCV.py --gui=False --config file.cfg
```

if a parameter file is available, or use command line options such as

```
simuCDCV.py --gui=False --demo='instant' --N0=10000 --N1=100000 \
--G0=500 --G1=500 --spec='[0.9]+[0.02]*5' --s=0.01 \
--mu='1e-4' --k=200
```

Note that parameters with `useDefault` set to `True` can be ignored if the default parameter is used. In addition, parameter `--optimized` could be used to load the optimized version of a `simuPOP` module. For this particular configuration, the optimized module is 30% faster (62s vs. 40s) than the standard module.

4. The simulation function could be imported to another script as follows

```
from simuCDCV import simuCDCV
simuCDCV(model='instant', N0=10000, N1=10000, G0=500, G1=500,
spec=[0.9]+[0.02]*5, s=0.01, mu=1e-4, k=200)
```

document



Abstract

simuPOP is a general-purpose individual-based 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. In contrast to competing applications that use command-line options or configuration files to direct the execution of a limited number of predefined evolutionary scenarios, users of simuPOP's scripting interface could make use of many of its unique features, such as customized chromosome types, arbitrary nonrandom mating schemes, virtual subpopulations, information fields and Python operators, to construct and study almost arbitrarily complex evolutionary scenarios.

simuPOP is provided as a number of Python modules, which consist of a large number of Python objects and functions, including population, mating schemes, operators (objects that manipulate populations) and simulators to coordinate the evolutionary processes. It is the users' responsibility to write a Python script to glue these pieces together and form a simulation. At a more user-friendly level, an increasing number of functions and scripts contributed by simuPOP users is available in the online simuPOP cookbook. They provide useful functions for different applications (e.g. load and manipulate HapMap samples, import and export files from another application) and allow users who are unfamiliar with simuPOP to perform a large number of simulations ranging from basic population genetics models to generating datasets under complex evolutionary scenarios.

This document provides complete references to all classes and functions of simuPOP and its utility modules. 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. All resources, including a pdf version of this guide and a mailing list can be found at the simuPOP homepage <http://simupop.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.

10.1 Individual, Population, pedigree and Simulator

10.1.1 class `GenoStruTrait`

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 both `Individual` and `Population` 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, which can either be shared by all loci or be specified for each locus.
- Names of loci and chromosomes.
- Names of information fields attached to each individual.

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

`GenoStruTrait` ()

A `GenoStruTrait` object is created with the construction of a `Population` object and cannot be initialized directly.

`absLocusIndex` (*chrom*, *locus*)

return the absolute index of locus *locus* on chromosome *chrom*. c.f. `chromLocusPair`.

alleleName (*allele*, *locus*=0)
return the name of allele *allele* at *locus* specified by the *alleleNames* parameter of the *Population* function. *locus* could be ignored if alleles at all loci share the same names. If the name of an allele is unspecified, its value ('0', '1', '2', etc) is returned.

alleleNames (*locus*=0)
return a list of allele names at locus given by the *alleleNames* parameter of the *Population* function. *locus* could be ignored if alleles at all loci share the same names. This list does not have to cover all possible allele states of a population so *alleleNames*() ['`*allele*'] might fail (use *alleleNames*() ['`*allele*'] instead).

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.

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

chromName (*chrom*)
return the name of a chromosome *chrom*.

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

chromType (*chrom*)
return the type of a chromosome *chrom* (CUSTOMIZED, AUTOSOME, CHROMOSOME_X, CHROMOSOME_Y or MITOCHONDRIAL).

chromTypes ()
return the type of all chromosomes (CUSTOMIZED, AUTOSOME, CHROMOSOME_X, CHROMOSOME_Y, or MITOCHONDRIAL).

indexesOfLoci (*loci*=ALL_AVAIL)
return the indexes of loci with positions *positions* (list of (chr, pos) pairs). Raise a *ValueError* if any of the loci cannot be found.

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.

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

lociDist (*locus1*, *locus2*)
Return the distance between loci *locus1* and *locus2* on the same chromosome. A negative value will be returned if *locus1* is after *locus2*.

lociNames ()
return the names of all loci specified by the *lociNames* parameter of the *Population* function. An empty list will be returned if *lociNames* was not specified.

lociPos ()
 return the positions of all loci, specified by the *lociPos* parameter 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. Note that empty strings are used for loci without name but you cannot lookup such loci using this function.

locusName (locus)
 return the name of locus *locus* specified by the *lociNames* parameter of the *Population* function. An empty string will be returned if no name has been given to locus *locus*.

locusPos (locus)
 return the position of locus *locus* specified by the *lociPos* parameter of the *Population* function.

numChrom ()
 return the number of chromosomes.

numLoci (chrom)
 return the number of loci on chromosome *chrom*.

numLoci ()
 return a list of the number of loci on all chromosomes.

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 #-ploidy where # is the ploidy number.

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

10.1.2 class Individual

class Individual

A *Population* consists of individuals with the same genotypic structure. An *Individual* object cannot be created independently, but references to individuals 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*), the *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 chromosome, 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, is not restricted.

Individual ()

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

affected ()

Return `True` if this individual is affected.

allele (*idx*, *ploidy*=-1, *chrom*=-1)

return the current allele at a locus, using its absolute index *idx*. If a ploidy *ploidy* and/or a chromosome indexes is given, *idx* is relative to the beginning of specified homologous copy of chromosomes (if *chrom*=-1) or the beginning of the specified homologous copy of specified chromosome (if *chrom* >= 0).

alleleChar (*idx*, *ploidy*=-1, *chrom*=-1)

return the name of allele(*idx*, *ploidy*, *chrom*). If *idx* is invalid (e.g. second homologous copy of chromosome Y), '_' is returned.

alleleLineage (*idx*, *ploidy*=-1, *chrom*=-1)

return the lineage of the allele at a locus, using its absolute index *idx*. If a ploidy *ploidy* and/or a chromosome indexes is given, *idx* is relative to the beginning of specified homologous copy of chromosomes (if *chrom*=-1) or the beginning of the specified homologous copy of specified chromosome (if *chrom* >= 0).

This function returns 0 for modules without lineage information.

__cmp__ (*rhs*)

a python function used to compare the individual objects

genotype (*ploidy*=ALL_AVAIL, *chroms*=ALL_AVAIL)

return an editable array (a `carray` object) that represents all alleles of an individual. If *ploidy* or *chroms* is given, only alleles on the specified chromosomes and homologous copy of chromosomes will be returned. If multiple chromosomes are specified, there should not be gaps between chromosomes. This function ignores type of chromosomes so it will return unused alleles for sex and mitochondrial chromosomes.

info (*field*)

Return the value of an information field *field* (by index or name). `ind.info(name)` is equivalent to `ind.name` although the function form allows the use of indexes of information fields.

lineage (*ploidy*=ALL_AVAIL, *chroms*=ALL_AVAIL)

return an editable array (a `carray_lineage` object) that represents the lineages of all alleles of an individual. If *ploidy* or *chroms* is given, only lineages on the specified chromosomes and homologous copy of chromosomes will be returned. If multiple chromosomes are specified, there should not be gaps between chromosomes. This function ignores type of chromosomes so it will return lineage of unused alleles for sex and mitochondrial chromosomes. A `None` object will be returned for modules without lineage information.

mutants (*ploidy*=ALL_AVAIL, *chroms*=ALL_AVAIL)

return an iterator that iterate through all mutants (non-zero alleles) of an individual. Each mutant is presented as a tuple of (index, value) where index is the index of mutant ranging from zero to `totNumLoci() * ploidy() - 1`, so you will have to adjust indexes to check multiple alleles at a locus. If *ploidy* or *chroms* is given, only alleles on the specified chromosomes and homologous copy of chromosomes will be iterated. If multiple chromosomes are specified, there should not be gaps between chromosomes. This function ignores type of chromosomes so it will return unused alleles for sex and mitochondrial chromosomes.

setAffected (*affected*)

set affection status to *affected* (True or False).

setAllele (*allele*, *idx*, *ploidy*=-1, *chrom*=-1)

set allele *allele* to a locus, using its absolute index *idx*. If a ploidy *ploidy* and/or a chromosome indexes are given, *idx* is relative to the beginning of specified homologous copy of chromosomes (if *chrom*=-1) or the beginning of the specified homologous copy of specified chromosome (if *chrom* >= 0).

setAlleleLineage (*lineage*, *idx*, *ploidy*=-1, *chrom*=-1)

set lineage *lineage* to an allele, using its absolute index *idx*. If a ploidy *ploidy* and/or a chromosome indexes are given, *idx* is relative to the beginning of specified homologous copy of chromosomes (if *chrom*=-1) or the beginning of the specified homologous copy of specified chromosome (if *chrom* >= 0). This function does nothing for modules without lineage information.

setGenotype (*geno*, *ploidy*=ALL_AVAIL, *chroms*=ALL_AVAIL)

Fill the genotype of an individual using a list of alleles *geno*. If parameters *ploidy* and/or *chroms* are specified, alleles will be copied to only all or specified chromosomes on selected homologous copies of chromosomes. *geno* will be reused if its length is less than number of alleles to be filled. This function ignores type of chromosomes so it will set genotype for unused alleles for sex and mitochondrial chromosomes.

setInfo (*value*, *field*)

set the value of an information field *field* (by index or name) to *value*. `ind.setInfo(value, field)` is equivalent to `ind.field = value` although the function form allows the use of indexes of information fields.

setLineage (*lineage*, *ploidy*=*ALL_AVAIL*, *chroms*=*ALL_AVAIL*)

Fill the lineage of an individual using a list of IDs *lineage*. If parameters *ploidy* and/or *chroms* are specified, lineages will be copied to only all or specified chromosomes on selected homologous copies of chromosomes. *lineage* will be reused if its length is less than number of allelic lineage to be filled. This function ignores type of chromosomes so it will set lineage to unused alleles for sex and mitochondrial chromosomes. It does nothing for modules without lineage information.

setSex (*sex*)

set individual sex to MALE or FEMALE.

sex ()

return the sex of an individual, 1 for male and 2 for female.

10.1.3 class Population

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 arrangement 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.

Population (*size*=[], *ploidy*=2, *loci*=[], *chromTypes*=[], *lociPos*=[], *ancGen*=0, *chromNames*=[], *alleleNames*=[], *lociNames*=[], *subPopNames*=[], *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 customized chromosomes 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. If there is only one chromosome, *numLoci* instead of [*numLoci*] can be used.

chromTypes A list that specifies the type of each chromosome, which can be AUTOSOME, CHROMOSOME_X, CHROMOSOME_Y, or CUSTOMIZED. 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 chromosomes is not supported. CUSTOMIZED chromosomes are special chromosomes whose inheritance patterns are undefined. They rely on user-defined functions and operators to be passed from parents to offspring. Multiple customized chromosomes have to be arranged consecutively.

lociPos Positions of all loci on all chromosome, as a list of float numbers. Default to 1, 2, ... etc on each chromosome. *lociPos* should be arranged chromosome by chromosome. If *lociPos* are not in order within a chromosome, they will be re-arranged along with corresponding *lociNames* (if specified).

ancGen 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 ' ' for all chromosomes.

alleleNames A list or a nested list of allele names. If a list of alleles is given, it will be used for all loci in this population. For example, *alleleNames*=('A', 'C', 'T', 'G') gives names A, C, T, and G to alleles 0, 1, 2, and 3 respectively. If a nested list of names is given, it should specify alleles names for all loci.

lociNames A list of names for each locus. It can be empty or a list of unique names for each locus. If loci are not specified in order, loci names will be rearranged according to their position on the chromosome.

subPopNames A list of subpopulation names. All subpopulations will have name ' ' if this parameter is not specified.

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

absIndIndex (*idx*, *subPop*)

return the absolute index of an individual *idx* in subpopulation *subPop*.

addChrom (*lociPos*, *lociNames*=[], *chromName*="", *alleleNames*=[], *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. Allele names could be specified for all loci (a list of names) or differently for each locus (a

nested list of names), using parameter *alleleNames*. Empty loci names will be used if *lociNames* is not specified. The newly added alleles will have zero lineage in modules with lineage information.

addChromFrom (*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. Chromosomes of *pop*, if named, should not conflict with names of existing chromosome. This function merges genotypes on the new chromosomes from population *pop* individual by individual.

addIndFrom (*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.

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

Add a list of information fields *fields* to a population and initialize their values to *init*. If an information field already exists, it will be re-initialized.

addLoci (*chrom*, *pos*, *lociNames*=[], *alleleNames*=[])

Insert loci *lociNames* at positions *pos* on chromosome *chrom*. These parameters should be lists of the same length, although *names* may be ignored, in which case empty strings will be assumed. Single-value input is allowed for parameter *chrom* and *pos* if only one locus is added. 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. Optionally, allele names could be specified either for all loci (a single list) or each loci (a nested list). This function returns indexes of the inserted loci. Newly inserted alleles will have zero lineage in modules with lineage information.

addLociFrom (*pop*, *byName*=False)

Add loci from population *pop*. By default, chromosomes are merged by index and names of merged chromosomes of population *pop* will be ignored (merge of two chromosomes with different names will yield a warning). If *byName* is set to *True*, chromosomes in *pop* will be merged to chromosomes with identical names. 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. Allele lineages are also copied from *pop* in modules with lineage information.

ancestor (*idx*, *gen*, *subPop*=[])

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 also *useAncestralGen*). If a valid *subPop* is specified, *index* is relative to that *subPop*. Virtual subpopulation is not supported. Note that a float *idx* is acceptable as long as it rounds closely to an integer.

ancestralGens ()

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

clone ()

Create a cloned copy of a population. Note that Python statement *pop1 = pop* only creates a reference to an existing population *pop*.

__cmp__ (*rhs*)

a python function used to compare the population objects

dvars (*subPop*=[])

Return a wrapper of Python dictionary returned by *vars* (*subPop*) so that dictionary keys can be accessed as attributes.

extractIndividuals (*indexes*=[], *IDs*=[], *idField*="ind_id", *filter*=None)

Extract individuals with given absolute indexes (parameter *indexes*), IDs (parameter *IDs*, stored in infor-

mation field *idField*, default to *ind_id*), or a filter function (parameter *filter*). If a list of absolute indexes are specified, the present generation will be extracted and form a one-generational population. If a list of IDs are specified, this function will look through all ancestral generations and extract individuals with given ID. Individuals with shared IDs are allowed. In the last case, a user-defined Python function should be provided. This function should accept parameter "ind" or one or more of the information fields. All individuals, including ancestors if there are multiple ancestral generations, will be passed to this function. Individuals that returns *True* will be extracted. Extracted individuals will be in their original ancestral generations and subpopulations, even if some subpopulations or generations are empty. An *IndexError* will be raised if an index is out of bound but no error will be given if an invalid ID is encountered.

extractSubPops (*subPops*=*ALL_AVAIL*, *rearrange*=*False*)

Extract a list of (virtual) subpopulations from a population and create a new population. If *rearrange* is *False* (default), structure and names of extracted subpopulations are kept although extracted subpopulations can have fewer individuals if they are created from extracted virtual subpopulations. (e.g. it is possible to extract all male individuals from a subpopulation using a *SexSplitter()*). If *rearrange* is *True*, each (virtual) subpopulation in *subPops* becomes a new subpopulation in the extracted population in the order at which they are specified. Because each virtual subpopulation becomes a subpopulation, this function could be used, for example, to separate male and female individuals to two subpopulations (*subPops*=[(0,0) , (0,1)]). If overlapping (virtual) subpopulations are specified, individuals will be copied multiple times. This function only extract individuals from the present generation.

genotype (*subPop*=[])

Return an editable array of the genotype of all individuals in a population (if *subPop*=[], default), or individuals in a subpopulation *subPop*. Virtual subpopulation is unsupported.

indByID (*id*, *ancGens*=*ALL_AVAIL*, *idField*="ind_id")

Return a reference to individual with *id* stored in information field *idField* (default to *ind_id*). This function by default search the present and all ancestral generations (*ancGen*=*ALL_AVAIL*), but you can limit the search in specific generations if you know which generations to search (*ancGens*=[0,1] for present and parental generations) or *UNSPECIFIED* to search only the current generation. If no individual with *id* is found, an *IndexError* will be raised. A float *id* is acceptable as long as it rounds closely to an integer. Note that this function uses a dynamic searching algorithm which tends to be slow. If you need to look for multiple individuals from a static population, you might want to convert a population object to a pedigree object and use function *Pedigree.indByID*.

indInfo (*field*, *subPop*=[])

Return the values (as a list) of information field *field* (by index or name) of all individuals (if *subPop*=[], default), or individuals in a (virtual) subpopulation (if *subPop*=*sp* or (*sp*, *vsp*)).

individual (*idx*, *subPop*=[])

Return a reference to individual *idx* in the population (if *subPop*=[], default) or a subpopulation (if *subPop*=*sp*). Virtual subpopulation is not supported. Note that a float *idx* is acceptable as long as it rounds closely to an integer.

individuals (*subPop*=[])

Return an iterator that can be used to iterate through all individuals in a population (if *subPop*=[], default), or a (virtual) subpopulation (*subPop*=*spID* or (*spID*, *vspID*)). If you would like to iterate through multiple subpopulations in multiple ancestral generations, please use function *Population.allIndividuals()*.

lineage (*subPop*=[])

Return an editable array of the lineage of alleles for all individuals in a population (if *subPop*=[], default), or individuals in a subpopulation *subPop*. Virtual subpopulation is unsupported. **This function returns "None" for modules without lineage information.**

mergeSubPops (*subPops*=*ALL_AVAIL*, *name*="", *toSubPop*=-1)

Merge subpopulations *subPops*. If *subPops* is *ALL_AVAIL* (default), all subpopulations will be merged. *subPops* do not have to be adjacent to each other. They will all be merged to the subpopulation with the

smallest subpopulation ID, unless a subpopulation ID is specified using parameter `toSubPop`. Indexes of the rest of the subpopulation may be changed. A new name can be assigned to the merged subpopulation through parameter *name* (an empty *name* will be ignored). This function returns the ID of the merged subpopulation.

mutants (*subPop*=[])

Return an iterator that iterate through mutants of all individuals in a population (if `subPop=[]`, default), or individuals in a subpopulation *subPop*. Virtual subpopulation is unsupported. Each mutant is presented as a tuple of (index, value) where index is the index of mutant (from 0 to `totNumLoci()*ploidy()`) so you will have to adjust its value to check multiple alleles at a locus. This function ignores type of chromosomes so non-zero alleles in unused alleles of sex and mitochondrial chromosomes are also iterated.

numSubPop ()

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

numVirtualSubPop ()

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

popSize (*ancGen*=-1, *sex*=ANY_SEX)

Return the total number of individuals in all subpopulations of the current generation (default) or the ancestral generation *ancGen*. This function by default returns number of all individuals (`sex=ANY_SEX`), but it will return number of males (if `sex=MALE_ONLY`), number of females (if `sex=MALE_ONLY`), and number of male/female pairs (if `sex=PAIR_ONLY`) which is essentially less of the number of males and females.

push (*pop*)

Push population *pop* into the current population. Both populations should have the same genotypic structure. The current population is discarded if *ancestralDepth* (maximum number of ancestral generations to hold) is zero so no ancestral generation can be kept. Otherwise, the current population will become the parental generation of *pop*. If *ancGen* of a population is positive and there are already *ancGen* ancestral generations (c.f. `ancestralGens()`), the greatest ancestral generation will be discarded. In any case, `Population*pop*` becomes invalid as all its individuals are absorbed by the current population.

recodeAlleles (*alleles*, *loci*=ALL_AVAIL, *alleleNames*=[])

Recode alleles at *loci* (can be a list of loci indexes or names, or all loci in a population (`ALL_AVAIL`)) to other values according to parameter *alleles*. This parameter can a list of new allele numbers for alleles 0, 1, 2, ... (allele *x* will be recoded to `newAlleles[x]`, *x* outside of the range of *newAlleles* will not be recoded, although a warning will be given if `DBG_WARNING` is defined) or a Python function, which should accept one or both parameters *allele* (existing allele) and *locus* (index of locus). The return value will become the new allele. This function is intended to recode some alleles without listing all alleles in a list. It will be called once for each existing allele so it is not possible to recode an allele to different alleles. A new list of allele names could be specified for these *loci*. Different sets of names could be specified for each locus if a nested list of names are given. This function recode alleles for all subpopulations in all ancestral generations.

removeIndividuals (*indexes*=[], *IDs*=[], *idField*="ind_id", *filter*=None)

remove individual(s) by absolute indexes (parameter *index*) or their IDs (parameter *IDs*), or using a filter function (parameter *filter*). If *indexes* are used, only individuals at the current generation will be removed. If *IDs* are used, all individuals with one of the IDs at information field *idField* (default to "ind_id") will be removed. Although "ind_id" usually stores unique IDs of individuals, this function is frequently used to remove groups of individuals with the same value at an information field. An `IndexError` will be raised if an index is out of bound, but no error will be given if an invalid ID is specified. In the last case, a user-defined function should be provided. This function should accept parameter "ind" or one or more of the information fields. All individuals, including ancestors if there are multiple ancestral generations, will be passed to this function. Individuals that returns `True` will be removed. This function does not affect subpopulation structure in the sense that a subpopulation will be kept even if all individuals from it

are removed.

removeInfoFields (*fields*)

Remove information fields *fields* from a population.

removeLoci (*loci=UNSPECIFIED, keep=UNSPECIFIED*)

Remove *loci* (absolute indexes or names) 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 (virtual) subpopulation(s) *subPops* and all their individuals. This function can be used to remove complete subpopulations (with shifted subpopulation indexes) or individuals belonging to virtual subpopulations of a subpopulation. In the latter case, the subpopulations are kept even if all individuals have been removed. This function only handles the present generation.

resize (*sizes, propagate=False*)

Resize population by giving new subpopulation sizes *sizes*. 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.

save (*filename*)

Save population to a file *filename*, which can be loaded by a global function `loadPopulation(filename)`.

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). If there exists more than *depth* ancestral generations (if *depth* > 0), extra ancestral generations are removed.

setGenotype (*geno, subPop=[]*)

Fill the genotype of all individuals in a population (if *subPop*=[]) or in a (virtual) subpopulation *subPop* (if *subPop*=*sp* or (*sp, vsp*)) using a list of alleles *geno*. *geno* will be reused if its length is less than `subPopSize(subPop)*totNumLoci()*ploidy()`.

setIndInfo (*values, field, subPop=[]*)

Set information field *field* (specified by index or name) of all individuals (if *subPop*=[], default), or individuals in a (virtual) subpopulation (*subPop*=*sp* or (*sp, vsp*)) to *values*. *values* will be reused if its length is smaller than the size of the population or (virtual) subpopulation.

setInfoFields (*fields, init=0*)

Set information fields *fields* to a population and initialize them with value *init*. All existing information fields will be removed.

setLineage (*geno, subPop=[]*)

Fill the lineage of all individuals in a population (if *subPop*=[]) or in a (virtual) subpopulation *subPop* (if *subPop*=*sp* or (*sp, vsp*)) using a list of IDs *lineage*. *lineage* will be reused if its length is less than `subPopSize(subPop)*totNumLoci()*ploidy()`. This function returns directly for modules without lineage information.

setSubPopByIndInfo (*field*)

Rearrange individuals to their new subpopulations according to their integer values at information field *field* (value returned by `Individual::info(field)`). individuals with negative values at this *field* will be removed. Existing subpopulation names are kept. New subpopulations will have empty names.

setSubPopName (*name, subPop*)

Assign a name *name* to subpopulation *subPop*. Note that subpopulation names do not have to be unique.

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.

sortIndividuals (*infoFields*, *reverse=False*)

Sort individuals according to values at specified information fields (*infoFields*). Individuals will be sorted at an increasing order unless *reverse* is set to `true`.

splitSubPop (*subPop*, *sizes*, *names=[]*)

Split subpopulation *subPop* into subpopulations of given *sizes*, which should add up to the size of subpopulation *subPop* or 1, in which case *sizes* are treated as proportions. If *subPop* is not the last subpopulation, indexes of subpopulations after *subPop* are shifted. If *subPop* is named, the same name will be given to all new subpopulations unless a new set of *names* are specified for these subpopulations. This function returns the IDs of split subpopulations.

subPopBegin (*subPop*)

Return the index of the first individual in subpopulation *subPop*.

subPopByName (*name*)

Return the index of the first subpopulation with name *name*. An `IndexError` will be raised if subpopulations are not named, or if no subpopulation with name *name* is found. Virtual subpopulation name is not supported.

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*.

subPopName (*subPop*)

Return the “spName - vspName” (virtual named subpopulation), “” (unnamed non-virtual subpopulation), “spName” (named subpopulation) or “vspName” (unnamed virtual subpopulation), depending on whether subpopulation is named or if *subPop* is virtual.

subPopNames ()

Return the names of all subpopulations (excluding virtual subpopulations). An empty string will be returned for unnamed subpopulations.

subPopSizes (*ancGen=-1*)

Return the sizes of all subpopulations at the current generation (default) or specified ancestral generation *ancGen*. Virtual subpopulations are not considered.

swap (*rhs*)

Swap the content of two population objects, which can be handy in some particular circumstances. For example, you could swap out a population in a simulator.

updateInfoFieldsFrom (*fields*, *pop*, *fromFields=[]*, *ancGens=ALL_AVAIL*)

Update information fields *fields* from *fromFields* of another population (or Pedigree) *pop*. Two populations should have the same number of individuals. If *fromFields* is not specified, it is assumed to be the same as *fields*. If *ancGens* is not `ALL_AVAIL`, only the specified ancestral generations are updated.

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 afterward to restore the correct order of ancestral generations.

vars (*subPop*=[])
 return variables of a population as a Python dictionary. If a valid subpopulation *subPop* is specified, a dictionary `vars() ["subPop"] [subPop]` is returned. A `ValueError` will be raised if key *subPop* does not exist in `vars()`, or if key *subPop* does not exist in `vars() ["subPop"]`.

virtualSplitter ()
 Return the virtual splitter associated with the population, `None` will be returned if there is no splitter.

asPedigree (*idField*='ind_id', *fatherField*='father_id', *motherField*='mother_id')
 Convert the existing population object to a pedigree. After this function pedigree function should magically be usable for this function.

subPopSize (*subPop*=[], *ancGen*=-1, *sex*=ANY_SEX)
 Return the size of a subpopulation (`subPopSize(sp)`) or a virtual subpopulation (`subPopSize([sp, vsp])`) in the current generation (default) or a specified ancestral generation *ancGen*. If no *subpop* is given, it is the same as `popSize(ancGen, sex)`. Population and virtual subpopulation names can be used. This function by default returns number of all individuals (*sex*=ANY_SEX), but it will return number of males (if *sex*=MALE_ONLY), number of females (if *sex*=MALE_ONLY), and number of male/female pairs (if *sex*=PAIR_ONLY) which is essentially less of the number of males and females. `<group>2-subpopsize</group>lociList()`

allIndividuals (*subPops*=ALL_AVAIL, *ancGens*=True)
 Return an iterator that iterates through all (virtual) subpopulations in all ancestral generations. A list of (virtual) subpopulations (*subPops*) and a list of ancestral generations (*ancGens*, can be a single number) could be specified to iterate through only selected subpopulation and generations. Value ALL_AVAIL is acceptable in the specification of *sp* and/or *vsp* in specifying a virtual subpopulation (*sp*, *vsp*) for the iteration through all or specific virtual subpopulation in all or specific subpopulations.

evolve (*initOps*=[], *preOps*=[], *matingScheme*=MatingScheme(), *postOps*=[], *finalOps*=[], *gen*=-1, *dryrun*=False)
 Evolve the current population *gen* generations using mating scheme *matingScheme* and operators *initOps* (applied before evolution), *preOps* (applied to the parental population at the beginning of each life cycle), *postOps* (applied to the offspring population at the end of each life cycle) and *finalOps* (applied at the end of evolution). More specifically, this function creates a *Simulator* using the current population, call its *evolve* function using passed parameters and then replace the current population with the evolved population. Please refer to function `Simulator.evolve` for more details about each parameter.

10.1.4 class Pedigree

class Pedigree

The pedigree class is derived from the population class. Unlike a population class that emphasizes on individual properties, the pedigree class emphasizes on relationship between individuals. An unique ID for all individuals is needed to create a pedigree object from a population object. Compared to the *Population* class, a *Pedigree* object is optimized for access individuals by their IDs, regardless of population structure and ancestral generations. Note that the implementation of some algorithms rely on the fact that parental IDs are smaller than their offspring because individual IDs are assigned sequentially during evolution. Pedigrees with manually assigned IDs should try to obey such a rule.

Pedigree (*pop*, *loci*=[], *infoFields*=[], *ancGens*=ALL_AVAIL, *idField*="ind_id", *fatherField*="father_id", *motherField*="mother_id", *stealPop*=False)

Create a pedigree object from a population, using a subset of loci (parameter *loci*, can be a list of loci indexes, names, or ALL_AVAIL, default to no locus), information fields (parameter *infoFields*, default to no information field besides *idField*, *fatherField* and *motherField*), and ancestral generations (parameter *ancGens*, default to all ancestral generations). By default, information field *father_id* (parameter *fatherField*) and *mother_id* (parameter *motherField*) are used to locate parents identified by *ind_id* (parameter *idField*), which should store an unique ID for all individuals. Multiple individuals with the same

ID are allowed and will be considered as the same individual, but a warning will be given if they actually differ in genotype or information fields. Operators *IdTagger* and *PedigreeTagger* are usually used to assign such IDs, although function `sampling.indexToID` could be used to assign unique IDs and construct parental IDs from index based relationship recorded by operator *ParentsTagger*. A pedigree object could be constructed with one or no parent but certain functions such as relative tracking will not be available for such pedigrees. In case that you are no longer using your population object, you could steal the content from the population by setting *stealPop* to `True`.

clone()

Create a cloned copy of a Pedigree.

identifyAncestors (*IDs*=*ALL_AVAIL*, *subPops*=*ALL_AVAIL*, *ancGens*=*ALL_AVAIL*)

If a list of individuals (*IDs*) is given, this function traces backward in time and find all ancestors of these individuals. If *IDs* is *ALL_AVAIL*, ancestors of all individuals in the present generation will be located. If a list of (virtual) subpopulations (*subPops*) or ancestral generations (*ancGens*) is given, the search will be limited to individuals in these subpopulations and generations. This could be used to, for example, find all fathers of *IDs*. This function returns a list of IDs, which includes valid specified IDs. Invalid IDs will be silently ignored. Note that parameters *subPops* and *ancGens* will limit starting IDs if *IDs* is set to *ALL_AVAIL*, but specified IDs will not be trimmed according to these parameters.

identifyFamilies (*pedField*="", *subPops*=*ALL_AVAIL*, *ancGens*=*ALL_AVAIL*)

This function goes through all individuals in a pedigree and group related individuals into families. If an information field *pedField* is given, indexes of families will be assigned to this field of each family member. The return value is a list of family sizes corresponding to families 0, 1, 2, ... etc. If a list of (virtual) subpopulations (parameter *subPops*) or ancestral generations are specified (parameter *ancGens*), the search will be limited to individuals in these subpopulations and generations.

identifyOffspring (*IDs*=[], *subPops*=*ALL_AVAIL*, *ancGens*=*ALL_AVAIL*)

This function traces forward in time and find all offspring of individuals specified in parameter *IDs*. If a list of (virtual) subpopulations (*subPops*) or ancestral generations (*ancGens*) is given, the search will be limited to individuals in these subpopulations and generations. This could be used to, for example, find all male offspring of *IDs*. This function returns a list of IDs, which includes valid starting *IDs*. Invalid IDs are silently ignored. Note that parameters *subPops* and *ancGens* will limit search result but will not be used to trim specified *IDs*.

indByID (*id*)

Return a reference to individual with *id*. An `IndexError` will be raised if no individual with *id* is found. A float *id* is acceptable as long as it rounds closely to an integer.

individualsWithRelatives (*infoFields*, *sex*=[], *affectionStatus*=[], *subPops*=*ALL_AVAIL*, *ancGens*=*ALL_AVAIL*)

Return a list of IDs of individuals who have non-negative values at information fields *infoFields*. Additional requirements could be specified by parameters *sex* and *affectionStatus*. *sex* can be *ANY_SEX* (default), *MALE_ONLY*, *FEMALE_ONLY*, *SAME_SEX* or *OPPOSITE_SEX*, and *affectionStatus* can be *AFFECTED*, *UNAFFECTED* or *ANY_AFFECTION_STATUS* (default). This function by default check all individuals in all ancestral generations, but you could limit the search using parameter *subPops* (a list of (virtual) subpopulations) and ancestral generations *ancGens*. Relatives fall out of specified subpopulations and ancestral generations will be considered invalid.

locateRelatives (*relType*, *resultFields*=[], *sex*=*ANY_SEX*, *affectionStatus*=*ANY_AFFECTION_STATUS*, *ancGens*=*ALL_AVAIL*)

This function locates relatives (of type *relType*) of each individual and store their IDs in information fields *relFields*. The length of *relFields* determines how many relatives an individual can have.

Parameter *relType* specifies what type of relative to locate, which can be

- *SPOUSE* locate spouses with whom an individual has at least one common offspring.
- *OUTBRED_SPOUSE* locate non-sibling spouses, namely spouses with no shared parent.

- **OFFSPRING** all offspring of each individual.
- **COMMON_OFFSPRING** common offspring between each individual and its spouse (located by **SPOUSE** or **OUTBRED_SPOUSE**). *relFields* should consist of an information field for spouse and $m-1$ fields for offspring where m is the number of fields.
- **FULLSIBLING** siblings with common father and mother,
- **SIBLING** siblings with at least one common parent.

Optionally, you can specify the sex and affection status of relatives you would like to locate, using parameters *sex* and *affectionStatus*. *sex* can be **ANY_SEX** (default), **MALE_ONLY**, **FEMALE_ONLY**, **SAME_SEX** or **OPPOSITE_SEX**, and *affectionStatus* can be **AFFECTED**, **UNAFFECTED** or **ANY_AFFECTION_STATUS** (default). Only relatives with specified properties will be located.

This function will by default go through all ancestral generations and locate relatives for all individuals. This can be changed by setting parameter *ancGens* to certain ancestral generations you would like to process.

save (*filename*, *infoFields*=[], *loci*=[])

Save a pedigree to file *filename*. This function goes through all individuals of a pedigree and outputs in each line the ID of individual, IDs of his or her parents, sex ('M' or 'F'), affection status ('A' or 'U'), values of specified information fields *infoFields* and genotypes at specified loci (parameter *loci*, which can be a list of loci indexes, names, or **ALL_AVAIL**). Allele numbers, instead of their names are outputted. Two columns are used for each locus if the population is diploid. This file can be loaded using function *loadPedigree* although additional information such as names of information fields need to be specified. This format differs from a “.ped file used in some genetic analysis software in that there is no family ID and IDs of all individuals have to be unique. Note that parental IDs will be set to zero if the parent is not in the pedigree object. Therefore, the parents of individuals in the top-most ancestral generation will always be zero.

traceRelatives (*fieldPath*, *sex*=[], *affectionStatus*=[], *resultFields*=[], *ancGens*=**ALL_AVAIL**)

Trace a relative path in a population and record the result in the given information fields *resultFields*. This function is used to locate more distant relatives based on the relatives located by function *locateRelatives*. For example, after siblings and offspring of all individuals are located, you can locate mother's sibling's offspring using a *relative path*, and save their indexes in each individuals information fields *resultFields*.

A *relative path* consists of a *fieldPath* that specifies which information fields to look for at each step, a *sex* specifies sex choices at each generation, and a *affectionStatus* that specifies affection status at each generation. *fieldPath* should be a list of information fields, *sex* and *affectionStatus* are optional. If specified, they should be a list of **ANY_SEX**, **MALE_ONLY**, **FEMALE_ONLY**, **SAME_SEX** and **OppositeSex** for parameter *sex*, and a list of **UNAFFECTED**, **AFFECTED** and **ANY_AFFECTION_STATUS** for parameter *affectionStatus*.

For example, if *fieldPath* = [['father_id', 'mother_id'], ['sib1', 'sib2'], ['off1', 'off2']], and *sex* = [**ANY_SEX**, **MALE_ONLY**, **FEMALE_ONLY**], this function will locate *father_id* and *mother_id* for each individual, find all individuals referred by *father_id* and *mother_id*, find information fields *sib1* and *sib2* from these parents and locate male individuals referred by these two information fields. Finally, the information fields *off1* and *off2* from these siblings are located and are used to locate their female offspring. The results are father or mother's brother's daughters. Their indexes will be saved in each individuals information fields *resultFields*. If a list of ancestral generations is given in parameter *ancGens* is given, only individuals in these ancestral generations will be processed.

asPopulation ()

Convert the existing pedigree object to a population. This function will behave like a regular population after this function call.

10.1.5 class Simulator

class Simulator

A simuPOP simulator is responsible for evolving one or more populations forward in time, subject to various *operators*. Populations in a simulator are created from one or more replicates of specified populations. A number of functions are provided to access and manipulate populations, and most importantly, to evolve them.

Simulator (*pops*, *rep*=1, *stealPops*=True)

Create a simulator with *rep* (default to 1) replicates of populations *pops*, which is a list of populations although a single population object is also acceptable. Contents of passed populations are by default moved to the simulator to avoid duplication of potentially large population objects, leaving empty populations behind. This behavior can be changed by setting *stealPops* to False, in which case populations are copied to the simulator.

add (*pop*, *stealPop*=True)

Add a population *pop* to the end of an existing simulator. This function by default moves *pop* to the simulator, leaving an empty population for passed population object. If *steal* is set to False, the population will be copied to the simulator, and thus unchanged.

clone ()

Clone a simulator, along with all its populations. Note that Python assign statement `simul1 = simul` only creates a symbolic link to an existing simulator.

__cmp__ (*rhs*)

a Python function used to compare the simulator objects Note that mating schemes are not tested.

dvars (*rep*, *subPop*=[])

Return a wrapper of Python dictionary returned by `vars(rep, subPop)` so that dictionary keys can be accessed as attributes.

evolve (*initOps*=[], *preOps*=[], *matingScheme*=MatingScheme, *postOps*=[], *finalOps*=[], *gen*=-1, *dryrun*=False)

Evolve all populations *gen* generations, subject to several lists of operators which are applied at different stages of an evolutionary process. Operators *initOps* are applied to all populations (subject to applicability restrictions of the operators, imposed by the *rep* parameter of these operators) before evolution. They are used to initialize populations before evolution. Operators *finalOps* are applied to all populations after the evolution.

Operators *preOps*, and *postOps* are applied during the life cycle of each generation. These operators can be applied at all or some of the generations, to all or some of the evolving populations, depending the *begin*, *end*, *step*, *at* and *reps* parameters of these operators. These operators are applied in the order at which they are specified. populations in a simulator are evolved one by one. At each generation, operators *preOps* are applied to the parental generations. A mating scheme is then used to populate an offspring generation. For each offspring, his or her sex is determined before during- mating operators of the mating scheme are used to transmit parental genotypes. After an offspring generation is successfully generated and becomes the current generation, operators *postOps* are applied to the offspring generation. If any of the *preOps* and *postOps* fails (return False), the evolution of a population will be stopped. The generation number of a population, which is the variable "gen" in each populations local namespace, is increased by one if an offspring generation has been successfully populated even if a post-mating operator fails. Another variable "rep" will also be set to indicate the index of each population in the simulator. Note that populations in a simulator does not have to have the same generation number. You could reset a population's generation number by changing this variable.

Parameter *gen* can be set to a non-negative number, which is the number of generations to evolve. If a simulator starts at the beginning of a generation *g* (for example 0), a simulator will stop at the beginning (instead of the end) of generation *g + gen* (for example *gen*). If *gen* is negative (default), the evolution will continue indefinitely, until all replicates are stopped by operators that return False at some point (these operators are called *terminators*). At the end of the evolution, the generations that each replicates

have evolved are returned. Note that *finalOps* are applied to all applicable population, including those that have stopped before others.

If parameter *dryrun* is set to `True`, this function will print a description of the evolutionary process generated by function `describeEvolProcess()` and exits.

extract (*rep*)

Extract the *rep-th* population from a simulator. This will reduce the number of populations in this simulator by one.

numRep ()

Return the number of replicates.

population (*rep*)

Return a reference to the *rep-th* population of a simulator. The reference will become invalid once the simulator starts evolving or becomes invalid (removed). If an independent copy of the population is needed, you can use `population.clone()` to create a cloned copy or `simulator.extract()` to remove the population from the simulator.

populations ()

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

vars (*rep*, *subPop*=[*i*])

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

10.2 Virtual splitters

10.2.1 class BaseVspSplitter

class BaseVspSplitter

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.

Each VSP has a name. A default name is determined by each splitter but you can also assign a name to each VSP. The name of a VSP can be retrieved by function `BaseVspSplitter.name()` or `Population.subPopName()`.

Only one VSP splitter can be assigned to a population, which defined VSPs for all its subpopulations. If different splitters are needed for different subpopulations, a *CombinedSplitter* can be used.

BaseVspSplitter (*names*=[*i*])

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.

vspByName (*name*)

Return the index of a virtual subpopulation from its name. If multiple virtual subpopulations share the same name, the first *vsp* is returned.

10.2.2 class SexSplitter

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.

SexSplitter (*names=[]*)

Create a sex splitter that defines male and female VSPs. These VSPs are named `Male` and `Female` unless a new set of names are specified by parameter *names*.

name (*vsp*)

Return "Male" if *vsp*=0 and "Female" otherwise, unless a new set of names are specified.

numVirtualSubPop ()

Return 2.

10.2.3 class AffectionSplitter

class AffectionSplitter

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

AffectionSplitter (*names=[]*)

Create a splitter that defined two VSPs by affection status. These VSPs are named `Unaffected` and `Affected` unless a new set of names are specified by parameter *names*.

name (*vsp*)

Return "Unaffected" if *vsp*=0 and "Affected" if *vsp*=1, unless a new set of names are specified.

numVirtualSubPop ()

Return 2.

10.2.4 class InfoSplitter

class InfoSplitter

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

InfoSplitter (*field*, *values=[]*, *cutoff=[]*, *ranges=[]*, *names=[]*)

Create an information 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 $v < 1$, $1 \leq v < 2$ and $v \geq 2$ where *v* is the value of an individual at information field *field*. If parameter *ranges* is specified, each range defines a VSP. For example, *ranges*=[[1, 3], [2, 5]] defines two VSPs with $1 \leq v < 3$ and $2 \leq v < 5$. Of course, only one of the parameters *values*, *cutoff* and *ranges* should be defined, and values in *cutoff* should be distinct, and in an increasing order. A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

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), $v_1 \leq \text{field} < v_2$ and *field* $\geq v$ (the last VSP) if VSPs are defined by cutoff values. A user- specified name, if specified, will be returned instead.

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.

10.2.5 class ProportionSplitter

class ProportionSplitter

This splitter divides subpopulations into several VSPs by proportion.

ProportionSplitter (*proportions*=[], *names*=[])

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. A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

name (*vsp*)

Return the name of VSP *vsp*, which is "Prop p" where $p = \text{proportions}[\text{vsp}]$. A user specified name will be returned if specified.

numVirtualSubPop ()

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

10.2.6 class RangeSplitter

class RangeSplitter

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

RangeSplitter (*ranges*, *names*=[])

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 second VSP consists of individuals 40, 41, ..., 49. Note that a nested list has to be used even if only one range is defined. A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

name (*vsp*)

Return the name of VSP *vsp*, which is "Range [a, b]" where $[a, b]$ is range $\text{ranges}[\text{vsp}]$. A user specified name will be returned if specified.

numVirtualSubPop ()

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

10.2.7 class GenotypeSplitter

class GenotypeSplitter

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

GenotypeSplitter (*loci*, *alleles*, *phase=False*, *names*=[])

Create a splitter that defines VSPs by individual genotype at *loci* (can be indexes or names of one or more loci). 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* is 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, `loci=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 individuals 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, `loci=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 individuals in the second VSP having genotype `[2, 2]` at locus 1. If *phase* is set to `True`, the first VSP will only have 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.

A default set of names are given to each VSP unless a new set of names is given by parameter *names*.

name (*vsp*)

Return name of VSP *vsp*, which is "Genotype *loc1,loc2:genotype*" as defined by parameters *loci* and *alleles*. A user provided name will be returned if specified.

numVirtualSubPop ()

number of virtual subpops of subpopulation *sp*

10.2.8 class CombinedSplitter

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 become the fourth (index 3) and the fifth (index 4) VSPs of the combined splitter. In addition, a new set of VSPs could be defined as the union of one or more of the original VSPs. This splitter is usually used to define different types of VSPs to a population.

CombinedSplitter (*splitters=[]*, *vspMap=[]*, *names=[]*)

Create a combined splitter using a list of *splitters*. For example, `CombinedSplitter([SexSplitter(), AffectionSplitter()])` defines a combined splitter with four VSPs, defined by male (*vsp* 0), female (*vsp* 1), unaffected (*vsp* 2) and affected individuals (*vsp* 3). Optionally, a new set of VSPs could be defined by parameter *vspMap*. Each item in this parameter is a list of VSPs that will be combined to a single VSP. For example, `vspMap=[(0, 2), (1, 3)]` in the previous example will define two VSPs defined by male or unaffected, and female or affected individuals. VSP names are usually determined by splitters, but can also be specified using parameter *names*.

name (*vsp*)

Return the name of a VSP *vsp*, which is the name a VSP defined by one of the combined splitters unless a new set of names is specified. If a *vspMap* was used, names from different VSPs will be joined by "or".

numVirtualSubPop ()

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

10.2.9 class ProductSplitter

class ProductSplitter

This splitter takes several splitters and take their intersections as new VSPs. For example, if the first splitter defines 3 VSPs and the second splitter defines 2, 6 VSPs will be defined by splitting 3 VSPs defined by the first splitter each to two VSPs. This splitter is usually used to define finer VSPs from existing VSPs.

ProductSplitter (*splitters=[]*, *names=[]*)

Create a product splitter using a list of *splitters*. For example, `ProductSplitter([SexSplitter(), AffectionSplitter()])` defines four VSPs by male unaffected, male affected, female unaffected, and female affected individuals. VSP names are usually determined by splitters, but can also be specified using parameter *names*.

name (*vsp*)

Return the name of a VSP *vsp*, which is the names of individual VSPs separated by a comma, unless a new set of names is specified for each VSP.

numVirtualSubPop ()

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

10.3 Mating Schemes

10.3.1 class MatingScheme

class MatingScheme

This mating scheme is the base class of all mating schemes. It evolves a population generation by generation but does not actually transmit genotype.

MatingScheme (*subPopSize=[]*)

Create a base mating scheme that evolves a population without transmitting genotypes. At each generation, this mating scheme creates an offspring generation according to parameter *subPopSize*, which can be a list of subpopulation sizes (or a number if there is only one subpopulation) or a Python function which will be called at each generation, just before mating, to determine the subpopulation sizes of the offspring generation. The function should be defined with one or both parameters of *gen* and *pop* where *gen* is the current generation number and *pop* is the parental population just before mating. The return value of this function should be a list of subpopulation sizes for the offspring generation. A single number can be returned if there is only one subpopulation. The passed parental population is usually used to determine offspring population size from parental population size but you can also modify this population to prepare for mating. A common practice is to split and merge parental populations in this function so that you demographic related information and actions could be implemented in the same function.

10.3.2 class HomoMating

class HomoMating

A homogeneous mating scheme that uses a parent chooser to choose parents from a prental generation, and an offspring generator to generate offspring from chosen parents. It can be either used directly, or within a heterogeneous mating scheme. In the latter case, it can be applied to a (virtual) subpopulation.

HomoMating (*chooser, generator, subPopSize=[], subPops=ALL_AVAIL, weight=0*)

Create a homogeneous mating scheme using a parent chooser *chooser* and an offspring generator *generator*.

If this mating scheme is used directly in a simulator, it will be responsible for creating an offspring population according to parameter *subPopSize*. This parameter can be a list of subpopulation sizes (or a number if there is only one subpopulation) or a Python function which will be called at each generation to determine the subpopulation sizes of the offspring generation. Please refer to class [MatingScheme](#) for details about this parameter.

If this mating shcme is used within a heterogeneous mating scheme. Parameters *subPops* and *weight* are used to determine which (virtual) subpopulations this mating scheme will be applied to, and how many offspring this mating scheme will produce. Please refer to mating scheme [HeteroMating](#) for the use of these two parameters.

10.3.3 class HeteroMating

class HeteroMating

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

HeteroMating (*matingSchemes*, *subPopSize*=[], *shuffleOffspring*=True, *weightBy*=ANY_SEX)

Create a heterogeneous mating scheme that will apply a list of homogeneous mating schemes *matingSchemes* to different (virtual) subpopulations. The size of the offspring generation is determined by parameter *subPopSize*, which can be a list of subpopulation sizes or a Python function that returns a list of subpopulation sizes at each generation. Please refer to class *MatingScheme* for a detailed explanation of this parameter.

Each mating scheme defined in *matingSchemes* can be applied to one or more (virtual) subpopulation. If parameter *subPops* is not specified, a mating scheme will be applied to all subpopulations. If a list of (virtual) subpopulation is specified, the mating scheme will be applied to specific (virtual) subpopulations.

If multiple mating schemes are applied to the same subpopulation, a weight (parameter *weight*) can be given to each mating scheme to determine how many offspring it will produce. The default weight for all mating schemes are 0. In this case, the number of offspring each mating scheme produces is proportional to the number of individuals in its parental (virtual) subpopulation (default to all parents, but can be father for *weightBy*=MALE_ONLY, mother for *weightBy*=FEMALE_ONLY, or father mother pairs (less of number of father and mothers) for *weightBy*=PAIR_ONLY). If all weights are negative, the numbers of offspring are determined by the multiplication of the absolute values of the weights and their respective parental (virtual) subpopulation sizes. If all weights are positive, the number of offspring produced by each mating scheme is proportional to these weights, except for mating schemes with zero parental population size (or no father, no mother, or no pairs, depending on value of parameter *weightBy*). Mating schemes with zero weight in this case will produce no offspring. If both negative and positive weights are present, negative weights are processed before positive ones.

A sexual mating scheme might fail if a parental (virtual) subpopulation has no father or mother. In this case, you can set *weightBy* to PAIR_ONLY so a (virtual) subpopulation will appear to have zero size, and will thus contribute no offspring to the offspring population. Note that the perceived parental (virtual) subpopulation size in this mode (and in modes of MALE_ONLY, FEMALE_ONLY) during the calculation of the size of the offspring subpopulation will be roughly half of the actual population size so you might have to use *weight*=-2 if you would like to have an offspring subpopulation that is roughly the same size of the parental (virtual) subpopulation.

If multiple mating schemes are applied to the same subpopulation, offspring produced by these mating schemes are shuffled randomly. If this is not desired, you can turn off offspring shuffling by setting parameter *shuffleOffspring* to False.

10.3.4 class ConditionalMating

class ConditionalMating

A conditional mating scheme that applies different mating schemes according to a condition (similar to operator IfElse). The condition can be a fixed condition, an expression or a user-defined function, to determine which mating scheme to be used.

ConditionalMating (*cond*, *ifMatingScheme*, *elseMatingScheme*)

Create a conditional mating scheme that applies mating scheme *ifMatingScheme* if the condition *cond* is True, or *elseMatingScheme* if *cond* is False. If a Python expression (a string) is given to parameter *cond*, the expression will be evaluated in parental population's local namespace. When a Python function is specified, it accepts parameter *pop* for the parental population. The return value of this function should be True or False. Otherwise, parameter *cond* will be treated as a fixed condition (converted to True or False) upon which *ifMatingScheme* or *elseMatingScheme* will always be applied.

10.3.5 class PedigreeMating

class PedigreeMating

This mating scheme evolves a population following an existing pedigree structure. If the *Pedigree* object

has N ancestral generations and a present generation, it can be used to evolve a population for N generations, starting from the topmost ancestral generation. At the k -th generation, this mating scheme produces an offspring generation according to subpopulation structure of the $N-k-1$ ancestral generation in the pedigree object (e.g. producing the offspring population of generation 0 according to the $N-1$ ancestral generation of the pedigree object). For each offspring, this mating scheme copies individual ID and sex from the corresponding individual in the pedigree object. It then locates the parents of each offspring using their IDs in the pedigree object. A list of during mating operators are then used to transmit parental genotype to the offspring. The population being evolved must have an information field 'ind_id'.

PedigreeMating (*ped*, *ops*, *idField*="ind_id")

Creates a pedigree mating scheme that evolves a population according to *Pedigree* object *ped*. The evolved population should contain individuals with ID (at information field *idField*, default to 'ind_id') that match those individual in the topmost ancestral generation who have offspring. After parents of each individuals are determined from their IDs, a list of during-mating operators *ops* are applied to transmit genotypes. The return value of these operators are not checked.

parallelizable ()

FIXME: No document

10.3.6 class SequentialParentChooser

class SequentialParentChooser

This parent chooser chooses a parent from a parental (virtual) subpopulation sequentially. Natural selection is not considered. If the last parent is reached, this parent chooser will restart from the beginning of the (virtual) subpopulation.

SequentialParentChooser (*sexChoice*=ANY_SEX)

Create a parent chooser that chooses a parent from a parental (virtual) subpopulation sequentially. Parameter *choice* can be ANY_SEX (default), MALE_ONLY and FEMALE_ONLY. In the latter two cases, only male or female individuals are selected. A `RuntimeError` will be raised if there is no male or female individual from the population.

chooseParents ()

Return chosen parents from a population if the parent chooser object is created with a population.

initialize (*pop*, *subPop*)

Initialize a parent chooser for subpopulation *subPop* of population *pop*.

10.3.7 class SequentialParentsChooser

class SequentialParentsChooser

This parent chooser chooses two parents (a father and a mother) sequentially from their respective sex groups. Selection is not considered. If all fathers (or mothers) are exhausted, this parent chooser will choose fathers (or mothers) from the beginning of the (virtual) subpopulation again.

SequentialParentsChooser ()

Create a parent chooser that chooses two parents sequentially from a parental (virtual) subpopulation.

10.3.8 class RandomParentChooser

class RandomParentChooser

This parent chooser chooses a parent randomly from a (virtual) parental subpopulation. Parents are chosen with or without replacement. If parents are chosen with replacement, a parent can be selected multiple times. If individual fitness values are assigned to individuals (stored in an information field *selectionField* (default to

"fitness"), individuals will be chosen at a probability proportional to his or her fitness value. If parents are chosen without replacement, a parent can be chosen only once. An `RuntimeError` will be raised if all parents are exhausted. Natural selection is disabled in the without- replacement case.

RandomParentChooser (*replacement=True, selectionField="fitness", sexChoice=ANY_SEX*)

Create a random parent chooser that choose parents with or without replacement (parameter *replacement*, default to `True`). If selection is enabled and information field *selectionField* exists in the passed population, the probability that a parent is chosen is proportional to his/her fitness value stored in *selectionField*. This parent chooser by default chooses parent from all individuals (`ANY_SEX`), but it can be made to select only male (`MALE_ONLY`) or female (`FEMALE_ONLY`) individuals by setting parameter *sexChoice*.

chooseParents ()

Return chosen parents from a population if the parent chooser object is created with a population.

initialize (*pop, subPop*)

Initialize a parent chooser for subpopulation *subPop* of population *pop*.

10.3.9 class RandomParentsChooser

class RandomParentsChooser

This parent chooser chooses two parents, a male and a female, randomly from a (virtual) parental subpopulation. Parents are chosen with or without replacement from their respective sex group. If parents are chosen with replacement, a parent can be selected multiple times. If individual fitness values are assigned (stored in information field *selectionField*, default to "fitness", the probability that an individual is chosen is proportional to his/her fitness value among all individuals with the same sex. If parents are chosen without replacement, a parent can be chosen only once. An `RuntimeError` will be raised if all males or females are exhausted. Natural selection is disabled in the without-replacement case.

RandomParentsChooser (*replacement=True, selectionField="fitness"*)

Create a random parents chooser that choose two parents with or without replacement (parameter *replacement*, default to `True`). If selection is enabled and information field *selectionField* exists in the passed population, the probability that a parent is chosen is proportional to his/her fitness value stored in *selectionField*.

chooseParents ()

Return chosen parents from a population if the parent chooser object is created with a population.

initialize (*pop, subPop*)

Initialize a parent chooser for subpopulation *subPop* of population *pop*.

10.3.10 class PolyParentsChooser

class PolyParentsChooser

This parent chooser is similar to random parents chooser but instead of selecting a new pair of parents each time, one of the parents in this parent chooser will mate with several spouses before he/she is replaced. This mimicks multi-spouse mating schemes such as polygyny or polyandry in some populations. Natural selection is supported for both sexes.

PolyParentsChooser (*polySex=MALE, polyNum=1, selectionField="fitness"*)

Create a multi-spouse parents chooser where each father (if *polySex* is `MALE`) or mother (if *polySex* is `FEMALE`) has *polyNum* spouses. The parents are chosen with replacement. If individual fitness values are assigned (stored to information field *selectionField*, default to "fitness"), the probability that an individual is chosen is proportional to his/her fitness value among all individuals with the same sex.

chooseParents ()

Return chosen parents from a population if the parent chooser object is created with a population.

initialize (*pop*, *subPop*)

Initialize a parent chooser for subpopulation *subPop* of *population* *pop*.

10.3.11 class CombinedParentsChooser

class CombinedParentsChooser

This parent chooser accepts two parent choosers. It takes one parent from each parent chooser and return them as father and mother. Because two parent choosers do not have to choose parents from the same virtual subpopulation, this parent chooser allows you to choose parents from different subpopulations.

CombinedParentsChooser (*fatherChooser*, *motherChooser*, *allowSelfing=True*)

Create a Python parent chooser using two parent choosers *fatherChooser* and *motherChooser*. It takes one parent from each parent chooser and return them as father and mother. If two valid parents are returned, the first valid parent (father) will be used for *fatherChooser*, the second valid parent (mother) will be used for *motherChooser*. Although these two parent choosers are supposed to return a father and a mother respectively, the sex of returned parents are not checked so it is possible to return parents with the same sex using this parents chooser. This choose by default allows the selection of the same parents as father and mother (self-fertilization), unless a parameter *allowSelfing* is used to disable it.

chooseParents ()

Return chosen parents from a population if the parent chooser object is created with a population.

initialize (*pop*, *subPop*)

Initialize a parent chooser for subpopulation *subPop* of *population* *pop*.

10.3.12 class PyParentsChooser

class PyParentsChooser

This parent chooser accepts a Python generator function that repeatedly yields one or two parents, which can be references to individual objects or indexes relative to each subpopulation. The parent chooser calls the generator function with parental population and a subpopulation index for each subpopulation and retrieves parents repeatedly using the iterator interface of the generator function.

This parent chooser does not support virtual subpopulation directly. However, because virtual subpopulations are defined in the passed parental population, it is easy to return parents from a particular virtual subpopulation using virtual subpopulation related functions.

PyParentsChooser (*generator*)

Create a Python parent chooser using a Python generator function *parentsGenerator*. This function should accept one or both of parameters *pop* (the parental population) and *subPop* (index of subpopulation) and return the reference or index (relative to subpopulation) of a parent or a pair of parents repeatedly using the iterator interface of the generator function.

chooseParents ()

Return chosen parents from a population if the parent chooser object is created with a population.

initialize (*pop*, *subPop*)

Initialize a parent chooser for subpopulation *subPop* of *population* *pop*.

10.3.13 class OffspringGenerator

class OffspringGenerator

An *offspring generator* generates offspring from parents chosen by a parent chooser. It is responsible for creating a certain number of offspring, determining their sex, and transmitting genotypes from parents to offspring.

OffspringGenerator (*ops*, *numOffspring*=1, *sexMode*=RANDOM_SEX)

Create a basic offspring generator. This offspring generator uses *ops* genotype transmitters to transmit genotypes from parents to offspring.

A number of *during-mating operators* (parameter *ops*) can be used to, among other possible duties such as setting information fields of offspring, transmit genotype from parents to offspring. This general offspring generator does not have any default during-mating operator but all stock mating schemes use an offspring generator with a default operator. For example, a `mendelianOffspringGenerator` is used by `RandomMating` to transmit genotypes. Note that applicability parameters `begin`, `step`, `end`, `at` and `reps` could be used in these operators but negative population and generation indexes are unsupported.

Parameter *numOffspring* is used to control the number of offspring per mating event, or in another word the number of offspring in each family. It can be a number, a Python function or generator, or a mode parameter followed by some optional arguments. If a number is given, given number of offspring will be generated at each mating event. If a Python function is given, it will be called each time when a mating event happens. When a generator function is specified, it will be called for each subpopulation to provide number of offspring for all mating events during the populating of this subpopulation. Current generation number will be passed to this function or generator function if parameter “gen” is used in this function. In the last case, a tuple (or a list) in one of the following forms can be given:

- (GEOMETRIC_DISTRIBUTION, *p*)
- (POISSON_DISTRIBUTION, *p*), *p* > 0
- (BINOMIAL_DISTRIBUTION, *p*, *N*), 0 < *p* ≤ 1, *N* > 0
- (UNIFORM_DISTRIBUTION, *a*, *b*), 0 ≤ *a* ≤ *b*.

In this case, the number of offspring will be determined randomly following the specified statistical distributions. Because families with zero offspring are silently ignored, the distribution of the observed number of offspring per mating event (excluding zero) follows zero-truncated versions of these distributions.

Parameter *numOffspring* specifies the number of offspring per mating event but the actual surviving offspring can be less than specified. More specifically, if any during-mating operator returns `False`, an offspring will be discarded so the actually number of offspring of a mating event will be reduced. This is essentially how during-mating selector works.

Parameter *sexMode* is used to control the sex of each offspring. Its default value is usually `RANDOM_SEX` which assign `MALE` or `FEMALE` to each individual randomly, with equal probabilities. If `NO_SEX` is given, offspring sex will not be changed. *sexMode* can also be one of

- (PROB_OF_MALES, *p*) where *p* is the probability of male for each offspring,
- (NUM_OF_MALES, *n*) where *n* is the number of males in a mating event. If *n* is greater than or equal to the number of offspring in this family, all offspring in this family will be `MALE`.
- (NUM_OF_FEMALES, *n*) where *n* is the number of females in a mating event,
- (SEQUENCE_OF_SEX, *s1*, *s2* ...) where *s1*, *s2* etc are `MALE` or `FEMALE`. The sequence will be used for each mating event. It will be reused if the number of offspring in a mating event is greater than the length of sequence.
- (GLOBAL_SEQUENCE_OF_SEX, *s1*, *s2*, ...) where *s1*, *s2* etc are `MALE` or `FEMALE`. The sequence will be used across mating events. It will be reused if the number of offspring in a subpopulation is greater than the length of sequence.

Finally, parameter *sexMode* accepts a function or a generator function. A function will be called whenever an offspring is produced. A generator will be created at each subpopulation and will be used to produce sex for all offspring in this subpopulation. No parameter is accepted.

10.3.14 class ControlledOffspringGenerator

class ControlledOffspringGenerator

This offspring generator populates an offspring population and controls allele frequencies at specified loci. At each generation, expected allele frequencies at these loci are passed from a user defined allele frequency *trajectory* function. The offspring population is populated in two steps. At the first step, only families with disease alleles are accepted until the expected number of disease alleles are met. At the second step, only families with wide type alleles are accepted to populate the rest of the offspring generation. This method is described in detail in “Peng et al, (2007) PLoS Genetics”.

ControlledOffspringGenerator (*loci*, *alleles*, *freqFunc*, *ops*=[], *numOffspring*=1, *sexMode*=RANDOM_SEX)

Create an offspring generator that selects offspring so that allele frequency at specified loci in the offspring generation reaches specified allele frequency. At the beginning of each generation, expected allele frequency of *alleles* at *loci* is returned from a user-defined trajectory function *freqFunc*. Parameter *loci* can be a list of loci indexes, names, or ALL_AVAIL. If there is no subpopulation, this function should return a list of frequencies for each locus. If there are multiple subpopulations, *freqFunc* can return a list of allele frequencies for all subpopulations or combined frequencies that ignore population structure. In the former case, allele frequencies should be arranged by loc0_sp0, loc1_sp0, ... loc0_sp1, loc1_sp1, ..., and so on. In the latter case, overall expected number of alleles are scattered to each subpopulation in proportion to existing number of alleles in each subpopulation, using a multinomial distribution.

After the expected alleles are calculated, this offspring generator accept and reject families according to their genotype at *loci* until allele frequencies reach their expected values. The rest of the offspring generation is then filled with families without only wild type alleles at these *loci*.

This offspring generator is derived from class *OffspringGenerator*. Please refer to class *OffspringGenerator* for a detailed description of parameters *ops*, *numOffspring* and *sexMode*.

10.4 Pre-defined mating schemes

10.4.1 class CloneMating

class CloneMating

A homogeneous mating scheme that uses a sequential parent chooser and a clone offspring generator.

CloneMating (*numOffspring*=1, *sexMode*=None, *ops*=CloneGenoTransmitter(), *subPopSize*=[], *subPops*=ALL_AVAIL, *weight*=0, *selectionField*=None)

Create a clonal mating scheme that clones parents to offspring using a *CloneGenoTransmitter*. Please refer to class *OffspringGenerator* for parameters *ops* and *numOffspring*, and to class *HomoMating* for parameters *subPopSize*, *subPops* and *weight*. Parameters *sexMode* and *selectionField* are ignored because this mating scheme does not support natural selection, and *CloneGenoTransmitter* copies sex from parents to offspring. Note that *CloneGenoTransmitter* by default also copies all parental information fields to offspring.

10.4.2 class RandomSelection

class RandomSelection

A homogeneous mating scheme that uses a random single-parent parent chooser with replacement, and a clone offspring generator. This mating scheme is usually used to simulate the basic haploid Wright-Fisher model but it can also be applied to diploid populations.

RandomSelection (*numOffspring=1, sexMode=None, ops=CloneGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Create a mating scheme that select a parent randomly and copy him or her to the offspring population. Please refer to class [RandomParentChooser](#) for parameter *selectionField*, to class [OffspringGenerator](#) for parameters *ops* and *numOffspring*, and to class [HomoMating](#) for parameters *subPopSize*, *subPops* and *weight*. Parameter *sexMode* is ignored because *cloneOffspringGenerator* copies sex from parents to offspring.

10.4.3 class RandomMating

class RandomMating

A homogeneous mating scheme that uses a random parents chooser with replacement and a Mendelian offspring generator. This mating scheme is widely used to simulate diploid sexual Wright-Fisher random mating.

RandomMating (*numOffspring=1, sexMode=RANDOM_SEX, ops=MendelianGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Creates a random mating scheme that selects two parents randomly and transmit genotypes according to Mendelian laws. Please refer to class [RandomParentsChooser](#) for parameter *selectionField*, to class [OffspringGenerator](#) for parameters *ops*, *sexMode* and *numOffspring*, and to class [HomoMating](#) for parameters *subPopSize*, *subPops* and *weight*.

10.4.4 class MonogamousMating

class MonogamousMating

A homogeneous mating scheme that uses a random parents chooser without replacement and a Mendelian offspring generator. It differs from the basic random mating scheme in that each parent can mate only once so there is no half-sibling in the population.

MonogamousMating (*numOffspring=1, sexMode=RANDOM_SEX, ops=MendelianGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField=None*)

Creates a monogamous mating scheme that selects each parent only once. Please refer to class [OffspringGenerator](#) for parameters *ops*, *sexMode* and *numOffspring*, and to class [HomoMating](#) for parameters *subPopSize*, *subPops* and *weight*. Parameter *selectionField* is ignored because this mating scheme does not support natural selection.

10.4.5 class PolygamousMating

class PolygamousMating

A homogeneous mating scheme that uses a multi-spouse parents chooser and a Mendelian offspring generator. It differs from the basic random mating scheme in that each parent of sex *polySex* will have *polyNum* spouses.

PolygamousMating (*polySex=MALE, polyNum=1, numOffspring=1, sexMode=RANDOM_SEX, ops=MendelianGenoTransmitter(), subPopSize=[], subPops=ALL_AVAIL, weight=0, selectionField='fitness'*)

Creates a polygamous mating scheme that each parent mates with multiple spouses. Please refer to class [PolyParentsChooser](#) for parameters *polySex*, *polyNum* and *selectionField*, to class [OffspringGenerator](#) for parameters *ops*, *sexMode* and *numOffspring*, and to class [HomoMating](#) for parameters *subPopSize*, *subPops* and *weight*.

10.4.6 class HaplodiploidMating

class HaplodiploidMating

A homogeneous mating scheme that uses a random parents chooser with replacement and a haplodiploid off-

spring generator. It should be used in a haplodiploid population where male individuals only have one set of homologous chromosomes.

HaplodiploidMating (*numOffspring=1.0*, *sexMode=RANDOM_SEX*,
ops=HaplodiploidGenoTransmitter(), *subPopSize=[]*, *subPops=ALL_AVAIL*,
weight=0, *selectionField='fitness'*)

Creates a mating scheme in haplodiploid populations. Please refer to class [RandomParentsChooser](#) for parameter *selectionField*, to class [OffspringGenerator](#) for parameters *ops*, *sexMode* and *numOffspring*, and to class [HomoMating](#) for parameters *subPopSize*, *subPops* and *weight*.

10.4.7 class SelfMating

class SelfMating

A homogeneous mating scheme that uses a random single-parent parent chooser with or without replacement (parameter *replacement*) and a selfing offspring generator. It is used to mimic self-fertilization in certain plant populations.

SelfMating (*replacement=True*, *numOffspring=1*, *sexMode=RANDOM_SEX*,
ops=SelfingGenoTransmitter(), *subPopSize=[]*, *subPops=ALL_AVAIL*, *weight=0*,
selectionField='fitness')

Creates a selfing mating scheme where two homologous copies of parental chromosomes are transmitted to offspring according to Mendelian laws. Please refer to class [RandomParentChooser](#) for parameter *replacement* and *selectionField*, to class [OffspringGenerator](#) for parameters *ops*, *sexMode* and *numOffspring*, and to class [HomoMating](#) for parameters *subPopSize*, *subPops* and *weight*.

10.4.8 class HermaphroditicMating

class HermaphroditicMating

A hermaphroditic mating scheme that chooses two parents randomly from the population regardless of sex. The parents could be chosen with or without replacement (parameter *replacement*). Selfing (if the same parents are chosen) is allowed unless *allowSelfing* is set to *False*

HermaphroditicMating (*replacement=True*, *allowSelfing=True*, *numOffspring=1*, *sex-*
Mode=RANDOM_SEX, *ops=MendelianGenoTransmitter()*, *subPop-*
Size=[], *subPops=ALL_AVAIL*, *weight=0*, *selectionField='fitness'*)

Creates a hermaphroditic mating scheme where individuals can serve as father or mother, or both (self-fertilization). Please refer to class [CombinedParentsChooser](#) for parameter *allowSelfing*, to `:class:'RandomParentChooser'` for parameter **replacement* and *selectionField*, to class [OffspringGenerator](#) for parameters *ops*, *sexMode* and *numOffspring*, and to class [HomoMating](#) for parameters *subPopSize*, *subPops* and *weight*.

10.4.9 class ControlledRandomMating

class ControlledRandomMating

A homogeneous mating scheme that uses a random sexual parents chooser with replacement and a controlled offspring generator using Mendelian genotype transmitter. It falls back to a regular random mating scheme if there is no locus to control or no trajectory is defined.

ControlledRandomMating (*loci=[]*, *alleles=[]*, *freqFunc=None*, *numOffspring=1*, *sex-*
Mode=RANDOM_SEX, *ops=MendelianGenoTransmitter()*, *subPop-*
Size=[], *subPops=ALL_AVAIL*, *weight=0*, *selectionField='fitness'*)

Creates a random mating scheme that controls allele frequency at loci *loci*. At each generation, function *freqFunc* will be called to obtain intended frequencies of alleles *alleles* at loci *loci*. The controlled offspring generator will control the acceptance of offspring so that the generation reaches desired

allele frequencies at these loci. If *loci* is empty or *freqFunc* is `None`, this mating scheme works identically to a `RandomMating` scheme. Rationals and applications of this mating scheme is described in details in a paper *Peng et al, 2007 (PLoS Genetics)*. Please refer to class `RandomParentsChooser` for parameters *selectionField*, to class `ControlledOffspringGenerator` for parameters *loci*, *alleles*, *freqFunc*, to class `OffspringGenerator` for parameters *ops*, *sexMode* and *numOffspring*, and to class `HomoMating` for parameters *subPopSize*, *subPops* and *weight*.

10.5 Utility Classes

10.5.1 class WithArgs

class WithArgs

This class wraps around a user-provided function and provides an attribute `args` so that simuPOP knows which parameters to send to the function. This is only needed if the function can not be defined with allowed parameters.

WithArgs (*func*, *args*)

Return a callable object that wraps around function *func*. Parameter *args* should be a list of parameter names.

10.5.2 class WithMode

class WithMode

This class wraps around a user-provided output string, function or file handle (acceptable by parameter `output` of operators) so that simuPOP knows which mode the output should be written to. For example, if the output of the operator is a binary compressed stream, `WithMode(output, 'b')` could be used to tell the operators to output bytes instead of string. This is most needed for Python 3 because files in Python 2 accepts string even if they are opened in binary mode.

WithMode (*output*, *mode=""*)

Return an object that wraps around *output* and tells simuPOP to output string in *mode*. This class currently only support *mode='t'* for text mode and *mode='b'* for binary output.

10.5.3 class RNG

class RNG

This random number generator class wraps around a number of random number generators from GNU Scientific Library. You can obtain and change the RNG used by the current simuPOP module through the `getRNG()` function, or create a separate random number generator and use it in your script.

RNG (*name=None*, *seed=0*)

Create a RNG object using specified name and seed. If *rng* is not given, environmental variable `GSL_RNG_TYPE` will be used if it is available. Otherwise, generator `mt19937` will be used. If *seed* is not given, `/dev/urandom`, `/dev/random`, or other system random number source will be used to guarantee that random seeds are used even if more than one simuPOP sessions are started simultaneously. Names of supported random number generators are available from `moduleInfo()['availableRNGs']`.

name ()

Return the name of the current random number generator.

randBinomial (*n*, *p*)

Generate a random number following a binomial distribution with parameters *n* and *p*.

randChisq (*nu*)

Generate a random number following a Chi-squared distribution with *nu* degrees of freedom.

randExponential (*mu*)

Generate a random number following a exponential distribution with parameter *mu*.

randGamma (*a*, *b*)

Generate a random number following a gamma distribution with a shape parameters *a* and scale parameter *b*.

randGeometric (*p*)

Generate a random number following a geometric distribution with parameter *p*.

randInt (*n*)

return a random number in the range of [0, 1, 2, ... n-1]

randMultinomial (*N*, *p*)

Generate a random number following a multinomial distribution with parameters *N* and *p* (a list of probabilities).

randNormal (*mu*, *sigma*)

Generate a random number following a normal distribution with mean *mu* and standard deviation *sigma*.

randPoisson (*mu*)

Generate a random number following a Poisson distribution with parameter *mu*.

randTruncatedBinomial (*n*, *p*)

Generate a positive random number following a zero-truncated binomial distribution with parameters *n* and *p*.

randTruncatedPoisson (*mu*)

Generate a positive random number following a zero-truncated Poisson distribution with parameter *mu*.

randUniform ()

Generate a random number following a rng_uniform [0, 1) distribution.

seed ()

Return the seed used to initialize the RNG. This can be used to repeat a previous session.

set (*name=None*, *seed=0*)

Replace the existing random number generator using RNG**name** with seed *seed*. If *seed* is 0, a random seed will be used. If *name* is empty, use the existing RNG but reset the seed.

10.5.4 class WeightedSampler

class WeightedSampler

A random number generator that returns 0, 1, ..., k-1 with probabilities that are proportional to their weights. For example, a weighted sampler with weights 4, 3, 2 and 1 will return numbers 0, 1, 2 and 3 with probabilities 0.4, 0.3, 0.2 and 0.1, respectively. If an additional parameter *N* is specified, the weighted sampler will return exact proportions of numbers if *N* numbers are returned. The version without additional parameter is similar to the `sample(prob, replace=FALSE)` function of the R statistical package.

WeightedSampler (*weights=[]*, *N=0*)

Creates a weighted sampler that returns 0, 1, ... k-1 when a list of *k* weights are specified (*weights*). *weights* do not have to add up to 1. If a non-zero *N* is specified, exact proportions of numbers will be returned in *N* returned numbers.

draw ()

Returns a random number between 0 and k-1 with probabilities that are proportional to specified weights.

drawSamples ($n=1$)
Returns a list of n random numbers

10.6 Global functions

10.6.1 Function closeOutput

closeOutput (*output=""*)

Output files specified by ' $>$ ' are closed immediately after they are written. Those specified by ' $>>$ ' and ' $>>>$ ' are closed by a simulator after `Simulator.evolve()`. However, these files will be kept open if the operators are applied directly to a population using the operators' function form. In this case, function `closeOutput` can be used to close a specific file *output*, and close all unclosed files if *output* is unspecified. An exception will be raised if *output* does not exist or it has already been closed.

10.6.2 Function describeEvolProcess

describeEvolProcess (*initOps=[]*, *preOps=[]*, *matingScheme=MatingScheme*, *postOps=[]*, *finalOps=[]*, *gen=-1*, *numRep=1*)

This function takes the same parameters as `Simulator.evolve` and output a description of how an evolutionary process will be executed. It is recommended that you call this function if you have any doubt how your simulation will proceed.

10.6.3 Function loadPopulation

loadPopulation (*file*)

load a population from a file saved by `Population::save()`.

10.6.4 Function loadPedigree

loadPedigree (*file*, *idField="ind_id"*, *fatherField="father_id"*, *motherField="mother_id"*, *ploidy=2*, *loci=[]*, *chromTypes=[]*, *lociPos=[]*, *chromNames=[]*, *alleleNames=[]*, *lociNames=[]*, *subPopNames=[]*, *infoFields=[]*)

Load a pedigree from a file saved by operator `PedigreeTagger` or function `Pedigree.save`. This file contains the ID of each offspring and their parent(s) and optionally sex ('M' or 'F'), affection status ('A' or 'U'), values of information fields and genotype at some loci. IDs of each individual and their parents are loaded to information fields *idField*, *fatherField* and *motherField*. Only numeric IDs are allowed, and individual IDs must be unique across all generations.

Because this file does not contain generation information, generations to which offspring belong are determined by the parent- offspring relationships. Individuals without parents are assumed to be in the top-most ancestral generation. This is the case for individuals in the top-most ancestral generation if the file is saved by function `Pedigree.save()`, and for individuals who only appear as another individual's parent, if the file is saved by operator `PedigreeTagger`. The order at which offspring is specified is not important because this function essentially creates a top-most ancestral generation using IDs without parents, and creates the next generation using offspring of these parents, and so on until all generations are recreated. That is to say, if you have a mixture of pedigrees with different generations, they will be lined up from the top most ancestral generation.

If individual sex is not specified, sex of of parents are determined by their parental roles (father or mother) but the sex of individuals in the last generation can not be determined so they will all be males. If additional information fields are given, their names have to be specified using parameter *infoFields*. The rest of the columns are assumed to be alleles, arranged *ploidy* consecutive columns for each locus. If parameter *loci* is not specified,

the number of loci is calculated by number of columns divided by *ploidy* (default to 2). All loci are assumed to be on one chromosome unless parameter *loci* is used to specified number of loci on each chromosome. Additional parameters such as *ploidy*, *chromTypes*, *lociPos*, *chromNames*, *alleleNames*, *lociNames* could be used to specified the genotype structured of the loaded pedigree. Please refer to class *Population* for details about these parameters.

10.6.5 Function moduleInfo

moduleInfo()

Return a dictionary with information regarding the currently loaded simuPOP module. This dictionary has the following keys:

- **revision:** revision number.
- **version:** simuPOP version string.
- **optimized:** Is this module optimized (True or False).
- **alleleType:** Allele type of the module (short, long or binary).
- **maxAllele:** the maximum allowed allele state, which is 1 for binary modules, 255 for short modules and 65535 for long modules.
- **compiler:** the compiler that compiles this module.
- **date:** date on which this module is compiled.
- **python:** version of python.
- **platform:** platform of the module.
- **wordsize:** size of word, can be either 32 or 64.
- **alleleBits:** the number of bits used to store an allele
- **maxNumSubPop:** maximum number of subpopulations.
- **maxIndex:** maximum index size (limits population size * total number of marker).
- **debug:** A dictionary with debugging codes as keys and the status of each debugging code (True or False) as their values.

10.6.6 Function getRNG

getRNG()

return the currently used random number generator

10.6.7 Function setRNG

setRNG(name="", seed=0)

Set random number generator. This function is obsolete but is provided for compatibility purposes. Please use *setOptions* instead

10.6.8 Function `setOptions`

setOptions (*numThreads=-1, name=None, seed=0*)

First argument is to set number of thread in openMP. The number of threads can be positive, integer (number of threads) or 0, which implies all available cores, or a number set by environmental variable `OMP_NUM_THREADS`. Second and third argument is to set the type or seed of existing random number generator using `RNG*name*` with *seed*. If using openMP, it sets the type or seed of random number generator of each thread.

10.6.9 Function `turnOnDebug`

turnOnDebug (*code=""*)

Set debug code *code*. More than one code could be specified using a comma separated string. Name of available codes are available from `moduleInfo()['debug'].keys()`.

10.6.10 Function `turnOffDebug`

turnOffDebug (*code="DBG_ALL"*)

Turn off debug code *code*. More than one code could be specified using a comma separated string. Default to turn off all debug codes.

11.1 Base class for all operators

11.1.1 class BaseOperator

class BaseOperator

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. In the latter case, operators are passed to the `evolve` function of a simulator, and are applied repeatedly during the evolution of the simulator.

The *BaseOperator* class is the base class for all operators. It defines a common user interface that specifies at which generations, at which stage of a life cycle, to which populations and subpopulations an operator is applied. These are achieved by a common set of parameters such as `begin`, `end`, `step`, `at`, `stage` for all operators. Note that a specific operator does not have to honor all these parameters. For example, a *Recombinator* can only be applied during mating so it ignores the `stage` parameter.

An operator can be applied to all or part of the generations during the evolution of a simulator. At the beginning of an evolution, a simulator is usually at the beginning of generation 0. If it evolves 10 generations, it evolves generations 0, 1, ..., and 9 (10 generations) and stops at the beginning of generation 10. A negative generation number `a` has generation number `10 + a`, with `-1` referring to the last evolved generation 9. Note that the starting generation number of a simulator can be changed by its `setGen()` member function.

Output from an operator is usually directed to the standard output (`sys.stdout`). This can be configured using an output specification string, which can be `' '` for no output, `'>'` standard terminal output (default), a filename prefixed by one or more `'>'` characters or a Python expression indicated by a leading exclamation mark (`'!expr'`). In the case of `'>filename'` (or equivalently `'filename'`), the output from an operator is written to this file. However, if two operators write to the same file `filename`, or if an operator writes to this file more than once, only the last write operation will succeed. In the case of `'>>filename'`, file `filename` will be opened at the beginning of the evolution and closed at the end. Outputs from multiple operators are appended. `>>>filename` works similar to `>>filename` but `filename`, if it already exists at the beginning of an evolutionary process, will not be cleared. If the output specification is prefixed by an exclamation mark, the string after the mark is considered as a Python expression. When an operator is applied to a population, this expression will be evaluated within the population's local namespace to obtain a population specific output

specification. As an advanced feature, a Python function can be assigned to this parameter. Output strings will be sent to this function for processing. Lastly, if the output stream only accept a binary output (e.g. a gzip stream), `WithMode(output, 'b')` should be used to let simuPOP convert string to bytes before writing to the output.

BaseOperator (*output, begin, end, step, at, reps, subPops, infoFields*)

The following parameters can be specified by all operators. However, an operator can ignore some parameters and the exact meaning of a parameter can vary.

output A string that specifies how output from an operator is written, which can be `' '` (no output), `'>'` (standard output), `'filename'` prefixed by one or more `'>'`, or an Python expression prefixed by an exclamation mark (`'!expr'`). If a file object, or any Python object with a `write` function is provided, the output will be write to this file. Alternatively, a Python function or a file object (any Python object with a `write` function) can be given which will be called with a string of output content. A global function `WithMode` can be used to let simuPOP output bytes instead of string.

begin The starting generation at which an operator will be applied. Default to 0. A negative number is interpreted as a generation counted from the end of an evolution (-1 being the last evolved generation).

end The last generation at which an operator will be applied. Default to -1, namely the last generation.

step The number of generations between applicable generations. Default to 1.

at A list of applicable generations. Parameters `begin`, `end`, and `step` will be ignored if this parameter is specified. A single generation number is also acceptable.

reps A list of applicable replicates. A common default value `ALL_AVAIL` is interpreted as all replicates in a simulator. Negative indexes such as -1 (last replicate) is acceptable. `rep=idx` can be used as a shortcut for `rep=[idx]`.

subPops A list of applicable (virtual) subpopulations, such as `subPops=[sp1, sp2, (sp2, vsp1)]`. `subPops=[sp1]` can be simplified as `subPops=sp1`. Negative indexes are not supported. A common default value (`ALL_AVAIL`) of this parameter represents all subpopulations of the population being applied. Support for this parameter vary from operator to operator and some operators do not support virtual subpopulations at all. Please refer to the reference manual of individual operators for their support for this parameter.

infoFields A list of information fields that will be used by an operator. You usually do not need to specify this parameter because operators that use information fields usually have default values for this parameter.

apply (*pop*)

Apply an operator to population *pop* directly, without checking its applicability.

clone ()

Return a cloned copy of an operator. This function is available to all operators.

11.2 Initialization

11.2.1 class InitSex

class InitSex

This operator initializes sex of individuals, either randomly or use a list of sexes.

InitSex (*maleFreq=0.5, maleProp=-1, sex=[], begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create an operator that initializes individual sex to MALE or FEMALE. By default, it assigns sex to individuals randomly, with equal probability of having a male or a female. This probability can be adjusted

through parameter *maleFreq* or be made to exact proportions by specifying parameter *maleProp*. Alternatively, a fixed sequence of sexes can be assigned. For example, if `sex=[MALE, FEMALE]`, individuals will be assigned MALE and FEMALE successively. Parameter *maleFreq* or *maleProp* are ignored if *sex* is given. If a list of (virtual) subpopulation is specified in parameter *subPop*, only individuals in these subpopulations will be initialized. Note that the *sex* sequence, if used, is assigned repeatedly regardless of (virtual) subpopulation boundaries so that you can assign *sex* to all individuals in a population.

11.2.2 class InitInfo

class InitInfo

This operator initializes given information fields with a sequence of values, or a user-provided function such as `random.random`.

InitInfo (*values*, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=[])

Create an operator that initialize individual information fields *infoFields* using a sequence of values or a user-defined function. If a list of values are given, it will be used sequentially for all individuals. The values will be reused if its length is less than the number of individuals. The values will be assigned repeatedly regardless of subpopulation boundaries. If a Python function is given, it will be called, without any argument, whenever a value is needed. If a list of (virtual) subpopulation is specified in parameter *subPop*, only individuals in these subpopulations will be initialized.

11.2.3 class InitGenotype

class InitGenotype

This operator assigns alleles at all or part of loci with given allele frequencies, proportions or values. This operator initializes all chromosomes, including unused genotype locations and customized chromosomes.

InitGenotype (*freq*=[], *genotype*=[], *prop*=[], *haplotypes*=[], *genotypes*=[], *loci*=ALL_AVAIL, *ploidy*=ALL_AVAIL, *begin*=0, *end*=1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=[])

This function creates an initializer that initializes individual genotypes with random alleles, genotypes, or haplotypes with specified frequencies (parameter *freq*) or proportions (parameter *prop*). If parameter *genotypes* or *haplotypes* is not specified, *freq* specifies the allele frequencies of alleles 0, 1, 2... respectively. Alternatively, you can use parameter *prop* to specified the exact proportions of alleles 0, 1, ..., although alleles with small proportions might not be assigned at all.

Values of parameter *prob* or *prop* should add up to 1. In addition to a vector, parameter *prob* and *prop* can also be a function that accepts optional parameters *loc*, *subPop* or *vsp* and returns a list of frequencies for alleles 0, 1, etc, or a number for frequency of allele 0 as a special case for each locus, subpopulation (parameter *subPop*), or virtual subpopulations (parameter *vsp*, pass as a tuple).

If parameter *genotypes* is specified, it should contain a list of genotypes (alleles on different strand of chromosomes) with length equal to population ploidy. Parameter *prob* and *prop* then specifies frequencies or proportions of each genotype, which can vary for each subpopulation but not each locus if the function form of parameters is used.

If parameter *haplotypes* is specified, it should contain a list of haplotypes (alleles on the same strand of chromosome) and parameter *prob* or *prop* specifies frequencies or proportions of each haplotype.

If *loci*, *ploidy* and/or *subPop* are specified, only specified loci, ploidy, and individuals in these (virtual) subpopulations will be initialized. Parameter *loci* can be a list of loci indexes, names or ALL_AVAIL. If the length of a haplotype is not enough to fill all loci, the haplotype will be reused. If a list (or a single) haplotypes are specified without *freq* or *prop*, they are used with equal probability.

In the last case, if a sequence of genotype is specified through parameter *genotype* (not *genotypes*), it will be used repeatedly to initialize all alleles sequentially. This works similar to function `Population.setGenotype()` except that you can limit the initialization to certain *loci* and *ploidy*.

11.2.4 class InitLineage

class InitLineage

This operator assigns lineages at all or part of loci with given values. This operator initializes all chromosomes, including unused lineage locations and customized chromosomes.

InitLineage (*lineage*=[], *mode*=PER_ALLELE, *loci*=ALL_AVAIL, *ploidy*=ALL_AVAIL, *begin*=0, *end*=1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"])

This function creates an initializer that initializes lineages with either a specified set of values or from the field *infoFields* (default to *ind_id*), whose value will be saved as the lineage of modified alleles. If a list of values is specified in parameter *lineage*, each value in this list is applied to one or more alleles so that each allele (PER_ALLELE, default mode), alleles on each chromosome (PER_CHROMOSOME), on chromosomes of each ploidy (PER_PLOIDY), or for each individual (PER_INDIVIDUAL) have the same lineage. A single value is allowed and values in *lineage* will be re-used if not enough values are provided. If an empty list is provided, values 1, 2, 3, .. will be used to provide a unique identify for each allele, genotype, chromosome, etc. If a valid field is specified (default to *ind_id*), the value of this field will be used for all alleles of each individual if *mode* is set to FROM_INFO, or be adjusted to produce positive values for alleles on the first ploidy, and negative values for the second ploidy (and so on) if *mode* equals to FROM_INFO_SIGNED. If *loci*, *ploidy* and/or *subPops* are specified, only specified loci, ploidy, and individuals in these (virtual) subpopulations will be initialized.

11.3 Expression and Statements

11.3.1 class PyOutput

class PyOutput

This operator outputs a given string when it is applied to a population.

PyOutput (*msg*="", *output*=">", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=[])

Creates a *PyOutput* operator that outputs a string *msg* to *output* (default to standard terminal output) when it is applied to a population. Please refer to class *BaseOperator* for a detailed description of common operator parameters such as *stage*, *begin* and *output*.

11.3.2 class PyEval

class PyEval

A *PyEval* operator evaluates a Python expression in a population's local namespace when it is applied to this population. The result is written to an output specified by parameter *output*.

PyEval (*expr*="", *stmts*="", *exposePop*="", *output*=">", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=Py_False, *infoFields*=[])

Create a *PyEval* operator that evaluates a Python expression *expr* in a population's local namespaces when it is applied to this population. This namespace can either be the population's local namespace (`pop.vars()`), or namespaces `subPop[sp]` for (virtual) subpop (`pop.vars(subpop)`) in specified *subPops*. If Python statements *stmts* is given (a single or multi-line string), the statement will be executed before *expr*. If *exposePop* is set to a non-empty string, the current population will be exposed in its

own local namespace as a variable with this name. This allows the execution of expressions such as `'pop.individual(0).allele(0)'`. The result of *expr* will be sent to an output stream specified by parameter *output*. The exposed population variable will be removed after *expr* is evaluated. Please refer to class *BaseOperator* for other parameters.

Note: Although the statements and expressions are evaluated in a population's local namespace, they have access to a global namespace which is the module global namespace. It is therefore possible to refer to any module variable in these expressions. Such mixed use of local and global variables is, however, strongly discouraged.

11.3.3 class PyExec

class PyExec

This operator executes given Python statements in a population's local namespace when it is applied to this population.

PyExec (*stmts=""*, *exposePop=""*, *output=">"*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=Py_False*, *infoFields=[]*)

Create a *PyExec* operator that executes statements *stmts* in a population's local namespace when it is applied to this population. This namespace can either be the population's local namespace (`pop.vars()`), or namespaces `subPop[sp]` for each (virtual) subpop (`pop.vars(subpop)`) in specified *subPops*. If *exposePop* is given, current population will be exposed in its local namespace as a variable named by *exposePop*. Although multiple statements can be executed, it is recommended that you use this operator to execute short statements and use *PyOperator* for more complex ones. Note that exposed population variables will be removed after the statements are executed.

11.3.4 class InfoEval

class InfoEval

Unlike operator *PyEval* and *PyExec* that work at the population level, in a population's local namespace, operator *InfoEval* works at the individual level, working with individual information fields. When this operator is applied to a population, information fields of eligible individuals are put into the local namespace of the population. A Python expression is then evaluated for each individual. The result is written to an output.

InfoEval (*expr=""*, *stmts=""*, *usePopVars=False*, *exposeInd=""*, *output=">"*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an operator that evaluate a Python expression *expr* using individual information fields and population variables as variables. If *exposeInd* is not empty, the individual itself will be exposed in the population's local namespace as a variable with name specified by *exposeInd*.

A Python expression (*expr*) is evaluated for each individual. The results are converted to strings and are written to an output specified by parameter *output*. Optionally, a statement (or several statements separated by newline) can be executed before *expr* is evaluated. The evaluation of this statement may change the value of information fields.

Parameter *usePopVars* is obsolete because population variables are always usable in such expressions.

11.3.5 class InfoExec

class InfoExec

Operator *InfoExec* is similar to *InfoEval* in that it works at the individual level, using individual informa-

tion fields as variables. This is usually used to change the value of information fields. For example, " $b=a*2$ " will set the value of information field b to $a*a$ for all individuals.

InfoExec (*stmts=""*, *usePopVars=False*, *exposeInd=""*, *output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an operator that executes Python statements *stmts* using individual information fields and population variables as variables. If *exposeInd* is not empty, the individual itself will be exposed in the population's local namespace as a variable with name specified by *exposeInd*.

One or more python statements (*stmts*) are executed for each individual. Information fields of these individuals are then updated from the corresponding variables. For example, $a=1$ will set information field a of all individuals to 1, $a=b$ will set information field a of all individuals to information field b or a population variable b if b is not an information field but a population variable, and $a=ind.sex()$ will set information field a of all individuals to its sex (needs *exposeInd='ind'*).

Parameter *usePopVars* is obsolete because population variables will always be usable.

11.4 Demographic models

11.4.1 class Migrator

class Migrator

This operator migrates individuals from (virtual) subpopulations to other subpopulations, according to either pre-specified destination subpopulation stored in an information field, or randomly according to a migration matrix.

In the former case, values in a specified information field (default to *migrate_to*) are considered as destination subpopulation for each individual. If *subPops* is given, only individuals in specified (virtual) subpopulations will be migrated where others will stay in their original subpopulation. Negative values are not allowed in this information field because they do not represent a valid destination subpopulation ID.

In the latter case, a migration matrix is used to randomly assign destination subpoulations to each individual. The elements in this matrix can be probabilities to migrate, proportions of individuals to migrate, or exact number of individuals to migrate.

By default, the migration matrix should have m by m elements if there are m subpopulations. Element (i, j) in this matrix represents migration probability, rate or count from subpopulation i to j . If *subPops* (length m) and/or *toSubPops* (length n) are given, the matrix should have m by n elements, corresponding to specified source and destination subpopulations. Subpopulations in *subPops* can be virtual subpopulations, which makes it possible to migrate, for example, males and females at different rates from a subpopulation. If a subpopulation in *toSubPops* does not exist, it will be created. In case that all individuals from a subpopulation are migrated, the empty subpopulation will be kept.

If migration is applied by probability, the row of the migration matrix corresponding to a source subpopulation is interpreted as probabilities to migrate to each destination subpopulation. Each individual's detination subpopulation is assigned randomly according to these probabilities. Note that the probability of staying at the present subpopulation is automatically calculated so the corresponding matrix elements are ignored.

If migration is applied by proportion, the row of the migration matrix corresponding to a source subpopulation is interpreted as proportions to migrate to each destination subpopulation. The number of migrants to each destination subpopulation is determined before random individuals are chosen to migrate.

If migration is applied by counts, the row of the migration matrix corresponding to a source subpopulation is interpreted as number of individuals to migrate to each detination subpopulation. The migrants are chosen randomly.

This operator goes through all source (virtual) subpopulations and assign destination subpopulation of each individual to an information field. Unexpected results may happen if individuals migrate from overlapping virtual subpopulations.

Migrator (*rate*=[], *mode*=BY_PROBABILITY, *toSubPops*=ALL_AVAIL, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["migrate_to"])

Create a Migrator that moves individuals from source (virtual) subpopulations *subPops* (default to migrate from all subpopulations) to destination subpopulations *toSubPops* (default to all subpopulations), according to existing values in an information field *infoFields**[0], or randomly according to a migration matrix **rate*. In the latter case, the size of the matrix should match the number of source and destination subpopulations.

Depending on the value of parameter *mode*, elements in the migration matrix (*rate*) are interpreted as either the probabilities to migrate from source to destination subpopulations (*mode* = BY_PROBABILITY), proportions of individuals in the source (virtual) subpopulations to the destination subpopulations (*mode* = BY_PROPORTION), numbers of migrants in the source (virtual) subpopulations (*mode* = BY_COUNTS), or ignored completely (*mode* = BY_IND_INFO). In the last case, parameter *subPops* is respected (only individuals in specified (virtual) subpopulations will migrate) but *toSubPops* is ignored.

Please refer to operator *BaseOperator* for a detailed explanation for all parameters.

11.4.2 class BackwardMigrator

class BackwardMigrator

This operator migrates individuals between all available or specified subpopulations, according to a backward migration matrix. It differs from *Migrator* in how migration matrixes are interpreted. Due to the limit of this model, this operator does not support migration by information field, migration by count (*mode* = BY_COUNT), migration from virtual subpopulations, migration between different number of subpopulations, and the creation of new subpopulation, as operator *Migrator* provides.

In contrast to a forward migration matrix where m_{ij} is considered the probability (proportion or count) of individuals migrating from subpopulation i to j , elements in a reverse migration matrix m_{ij} is considered the probability (proportion or count) of individuals migrating from subpopulation j to i , namely the probability (proportion or count) of individuals originates from subpopulation j .

If migration is applied by probability, the row of the migration matrix corresponding to a destination subpopulation is interpreted as probabilities to originate from each source subpopulation. Each individual's source subpopulation is assigned randomly according to these probabilities. Note that the probability of originating from the present subpopulation is automatically calculated so the corresponding matrix elements are ignored.

If migration is applied by proportion, the row of the migration matrix corresponding to a destination subpopulation is interpreted as proportions to originate from each source subpopulation. The number of migrants from each source subpopulation is determined before random individuals are chosen to migrate.

Unlike the forward migration matrix that describes how migration should be performed, the backward migration matrix describes the result of migration. The underlying forward migration matrix is calculated at each generation and is in theory not the same across generations.

This operator calculates the corresponding forward migration matrix from backward matrix and current population size. This process is not always feasible so an error will raise if no valid ending population size or forward migration matrix could be determined. Please refer to the simuPOP user's guide for an explanation of the theory behind forward and backward migration matrices.

BackwardMigrator (*rate*=[], *mode*=BY_PROBABILITY, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["migrate_to"])

Create a BackwardMigrator that moves individuals between *subPop* subpopulations randomly according to a backward migration matrix *rate*. The size of the matrix should match the number of subpopulations.

Depending on the value of parameter *mode*, elements in the migration matrix (*rate*) are interpreted as either the probabilities to originate from source subpopulations (*mode* = BY_PROBABILITY) or proportions of individuals originate from the source (virtual) subpopulations (*mode* = BY_PROPORTION). Migration by count is not supported by this operator.

Please refer to operator *BaseOperator* for a detailed explanation for all parameters.

11.4.3 class SplitSubPops

class SplitSubPops

Split a given list of subpopulations according to either sizes of the resulting subpopulations, proportion of individuals, or an information field. The resulting subpopulations will have the same name as the original subpopulation.

SplitSubPops (*subPops*=ALL_AVAIL, *sizes*=[], *proportions*=[], *names*=[], *randomize*=True, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *infoFields*=[])

Split a list of subpopulations *subPops* into finer subpopulations. A single subpopulation is acceptable but virtual subpopulations are not allowed. All subpopulations will be split if *subPops* is not specified.

The subpopulations can be split in three ways:

- If parameter *sizes* is given, each subpopulation will be split into subpopulations with given size. The *sizes* should add up to the size of all original subpopulations.
- If parameter *proportions* is given, each subpopulation will be split into subpopulations with corresponding proportion of individuals. *proportions* should add up to 1.
- If an information field is given (parameter *infoFields*), individuals having the same value at this information field will be grouped into a subpopulation. The number of resulting subpopulations is determined by the number of distinct values at this information field.

If parameter *randomize* is True (default), individuals will be randomized before a subpopulation is split. This is designed to remove artificial order of individuals introduced by, for example, some non-random mating schemes. Note that, however, the original individual order is not guaranteed even if this parameter is set to False.

Unless the last subpopulation is split, the indexes of existing subpopulations will be changed. If a subpopulation has a name, this name will become the name for all subpopulations separated from this subpopulation. Optionally, you can assign names to the new subpopulations using a list of names specified in parameter *names*. Because the same set of names will be used for all subpopulations, this parameter is not recommended when multiple subpopulations are split.

Please refer to operator *BaseOperator* for a detailed explanation for all parameters.

Note: Unlike operator *Migrator*, this operator does not require an information field such as *migrate_to*.

11.4.4 class MergeSubPops

class MergeSubPops

This operator merges subpopulations *subPops* to a single subpopulation. If *subPops* is ignored, all subpopulations will be merged. Virtual subpopulations are not allowed in *subPops*.

MergeSubPops (*subPops*=ALL_AVAIL, *name*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *infoFields*=[])

Create an operator that merges subpopulations *subPops* to a single subpopulation. If *subPops* is not given,

all subpopulations will be merged. The merged subpopulation will take the name of the first subpopulation being merged unless a new *name* is given.

Please refer to operator *BaseOperator* for a detailed explanation for all parameters.

11.4.5 class ResizeSubPops

class ResizeSubPops

This operator resizes subpopulations to specified sizes. individuals are added or removed depending on the new subpopulation sizes.

ResizeSubPops (*subPops=ALL_AVAIL, sizes=[], proportions=[], propagate=True, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, infoFields=[]*)

Resize given subpopulations *subPops* to new sizes *size*, or sizes proportional to original sizes (parameter *proportions*). All subpopulations will be resized if *subPops* is not specified. If the new size of a subpopulation is smaller than its original size, extra individuals will be removed. If the new size is larger, new individuals with empty genotype will be inserted, unless parameter *propagate* is set to *True* (default). In this case, existing individuals will be copied sequentially, and repeatedly if needed.

Please refer to operator *BaseOperator* for a detailed explanation for all parameters.

11.5 Genotype transmitters

11.5.1 class GenoTransmitter

class GenoTransmitter

This during mating operator is the base class of all genotype transmitters. It is made available to users because it provides a few member functions that can be used by derived transmitters, and by customized Python during mating operators.

GenoTransmitter (*output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a base genotype transmitter.

clearChromosome (*ind, ploidy, chrom*)

Clear (set alleles to zero) chromosome *chrom* on the *ploidy*-th homologous set of chromosomes of individual *ind*. It is equivalent to `ind.setGenotype([0], ploidy, chrom)`, except that it also clears allele lineage if it is executed in a module with lineage allele type.

copyChromosome (*parent, parPloidy, offspring, ploidy, chrom*)

Transmit chromosome *chrom* on the *parPloidy* set of homologous chromosomes from *parent* to the *ploidy* set of homologous chromosomes of *offspring*. It is equivalent to `offspring.setGenotype(parent.genotype(parPloidy, chrom), ploidy, chrom)`, except that it also copies allelic lineage when it is executed in a module with lineage allele type.

copyChromosomes (*parent, parPloidy, offspring, ploidy*)

Transmit the *parPloidy* set of homologous chromosomes from *parent* to the *ploidy* set of homologous chromosomes of *offspring*. Customized chromosomes are not copied. It is equivalent to `offspring.setGenotype(parent.genotype(parPloidy), ploidy)`, except that it also copies allelic lineage when it is executed in a module with lineage allele type.

11.5.2 class CloneGenoTransmitter

class CloneGenoTransmitter

This during mating operator copies parental genotype directly to offspring. This operator works for all mating schemes when one or two parents are involved. If both parents are passed, maternal genotype are copied. In addition to genotypes on all non-customized or specified chromosomes, sex and information fields are by default also copied from parent to offspring.

CloneGenoTransmitter (*output=""*, *chroms=ALL_AVAIL*, *begin=0*, *end=-1*, *step=1*, *at=[]*,
reps=ALL_AVAIL, *subPops=ALL_AVAIL*, *infoFields=ALL_AVAIL*)

Create a clone genotype transmitter (a during-mating operator) that copies genotypes from parents to offspring. If two parents are specified, genotypes are copied maternally. After genotype transmission, offspring sex and affection status is copied from the parent even if sex has been determined by an offspring generator. All or specified information fields (parameter *infoFields*, default to *ALL_AVAIL*) will also be copied from parent to offspring. Parameters *subPops* is ignored. This operator by default copies genotypes on all autosome and sex chromosomes (excluding customized chromosomes), unless a parameter *chroms* is used to specify which chromosomes to copy. This operator also copies allelic lineage when it is executed in a module with lineage allele type.

11.5.3 class MendelianGenoTransmitter

class MendelianGenoTransmitter

This Mendelian offspring generator accepts two parents and pass their genotypes to an offspring following Mendel's laws. Sex chromosomes are handled according to the sex of the offspring, which is usually determined in advance by an offspring generator. Customized chromosomes are not handled.

MendelianGenoTransmitter (*output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *sub-*
Pops=ALL_AVAIL, *infoFields=[]*)

Create a Mendelian genotype transmitter (a during-mating operator) that transmits genotypes from parents to offspring following Mendel's laws. Autosomes and sex chromosomes are handled but customized chromosomes are ignored. Parameters *subPops* and *infoFields* are ignored. This operator also copies allelic lineage when it is executed in a module with lineage allele type.

transmitGenotype (*parent*, *offspring*, *ploidy*)

Transmit genotype from parent to offspring, and fill the *ploidy* homologous set of chromosomes. This function does not set genotypes of customized chromosomes and handles sex chromosomes properly, according to offspring sex and *ploidy*.

11.5.4 class SelfingGenoTransmitter

class SelfingGenoTransmitter

A genotype transmitter (during-mating operator) that transmits parental genotype of a parent through self-fertilization. That is to say, the offspring genotype is formed according to Mendel's laws, only that a parent serves as both maternal and paternal parents.

SelfingGenoTransmitter (*output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *sub-*
Pops=ALL_AVAIL, *infoFields=[]*)

Create a self-fertilization genotype transmitter that transmits genotypes of a parent to an offspring through self-fertilization. Customized chromosomes are not handled. Parameters *subPops* and *infoFields* are ignored. This operator also copies allelic lineage when it is executed in a module with lineage allele type.

11.5.5 class HaplodiploidGenoTransmitter

class HaplodiploidGenoTransmitter

A genotype transmitter (during-mating operator) for haplodiploid populations. The female parent is considered as diploid and the male parent is considered as haploid (only the first homologous copy is valid). If the offspring is FEMALE, she will get a random copy of two homologous chromosomes of her mother, and get the only paternal copy from her father. If the offspring is MALE, he will only get a set of chromosomes from his mother.

HaplodiploidGenoTransmitter (*output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*,
subPops=ALL_AVAIL, *infoFields=[]*)

Create a haplodiploid genotype transmitter (during-mating operator) that transmit parental genotypes from parents to offspring in a haplodiploid population. Parameters *subPops* and *infoFields* are ignored. This operator also copies allelic lineage when it is executed in a module with lineage allele type.

11.5.6 class MitochondrialGenoTransmitter

class MitochondrialGenoTransmitter

This geno transmitter transmits the first homologous copy of a Mitochondrial chromosome. If no mitochondrial chromosome is present, it assumes that the first homologous copy of several (or all) Customized chromosomes are copies of mitochondrial chromosomes. This operator transmits the mitochondrial chromosome from the female parent to offspring for sexual reproduction, and any parent to offspring for asexual reproduction. If there are multiple chromosomes, the organelles are selected randomly. If this transmitter is applied to populations with more than one homologous copies of chromosomes, it transmits the first homologous copy of chromosomes and clears alleles (set to zero) on other homologous copies.

MitochondrialGenoTransmitter (*output=""*, *chroms=ALL_AVAIL*, *begin=0*, *end=-1*,
step=1, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*,
infoFields=[])

Create a mitochondrial genotype transmitter that treats the Mitochondrial chromosome, or Customized chromosomes if no Mitochondrial chromosome is specified, or a list of chromosomes specified by *chroms*, as human mitochondrial chromosomes. These chromosomes should have the same length and the same number of loci. This operator transmits these chromosomes randomly from the female parent to offspring of both sexes. It also copies allelic lineage when it is executed in a module with lineage allele type.

11.5.7 class Recombinator

class Recombinator

A genotype transmitter (during-mating operator) that transmits parental chromosomes to offspring, subject to recombination and gene conversion. This can be used to replace *MendelianGenoTransmitter* and *SelfingGenoTransmitter*. It does not work in haplodiploid populations, although a customized genotype transmitter that makes uses this operator could be defined. Please refer to the simuPOP user's guide or online cookbook for details.

Recombination could be applied to all adjacent markers or after specified loci. Recombination rate between two adjacent markers could be specified directly, or calculated using physical distance between them. In the latter case, a recombination intensity is multiplied by physical distance between markers.

Gene conversion is interpreted as double-recombination events. That is to say, if a recombination event happens, it has a certain probability (can be 1) to become a conversion event, namely triggering another recombination event down the chromosome. The length of the converted chromosome can be controlled in a number of ways.

Note: simuPOP does not assume any unit to loci positions so recombination intensity could be explained differently (e.g. cM/Mb, Morgan/Mb) depending on your interpretation of loci positions. For example, if basepair is used for loci position, *intensity=10^-8* indicates 10^{-8} per basepair, which is equivalent to 10^{-2}

per Mb or 1 cM/Mb. If Mb is used for physical positions, the same recombination intensity could be achieved by `intensity=0.01`.

Recombinator (*rates=[]*, *intensity=-1*, *loci=ALL_AVAIL*, *convMode=NO_CONVERSION*, *output=""*,
begin=0, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create a Recombinator (a mendelian genotype transmitter with recombination and gene conversion) that passes genotypes from parents (or a parent in case of self-fertilization) to offspring.

Recombination happens by default between all adjacent markers but can be limited to a given set of *loci*, which can be a list of loci indexes, names, list of chromosome position pairs, `ALL_AVAIL`, or a function with optional parameter `pop` that will be called at each generation to determine indexes of loci. Each locus in this list specifies a recombination point between the locus and the locus immediately **after** it. Loci that are the last locus on each chromosome are ignored.

If a single recombination rate (parameter *rates*) is specified, it will be used for all loci (all loci or loci specified by parameter *loci*), regardless of physical distances between adjacent loci.

If a list of recombination rates are specified in *rates*, different recombination rates could be applied after a list of specified loci (between loci and their immediate neighbor to the right). The loci should be specified by parameter *loci* as a list with the same length as *rates*, or `ALL_AVAIL` (default) in which case the length of *rates* should equal to the total number of loci. Note that recombination rates specified for the last locus on each chromosome are ignored because simuPOP assumes free recombination between chromosomes.

A recombination intensity (*intensity*) can be used to specify recombination rates that are proportional to physical distances between adjacent markers. If the physical distance between two markers is *d*, the recombination rate between them will be `intensity * d`. No unit is assumed for loci position and recombination intensity.

Gene conversion is controlled using parameter *convMode*, which can be

- `NoConversion`: no gene conversion (default).
- `(NUM_MARKERS, prob, n)`: With probability *prob*, convert a fixed number (*n*) of markers if a recombination event happens.
- `(GEOMETRIC_DISTRIBUTION, prob, p)`: With probability *prob*, convert a random number of markers if a recombination event happens. The number of markers converted follows a geometric distribution with probability *p*.
- `(TRACT_LENGTH, prob, n)`: With probability *prob*, convert a region of fixed tract length (*n*) if a recombination event happens. The actual number of markers converted depends on loci positions of surrounding loci. The starting position of this tract is the middle of two adjacent markers. For example, if four loci are located at 0, 1, 2, 3 respectively, a conversion event happens between 0 and 1, with a tract length 2 will start at 0.5 and end at 2.5, covering the second and third loci.
- `(EXPONENTIAL_DISTRIBUTION, prob, p)`: With probability *prob*, convert a region of random tract length if a recombination event happens. The distribution of tract length follows an exponential distribution with probability *p*. The actual number of markers converted depends on loci positions of surrounding loci.

simuPOP uses this probabilistic model of gene conversion because when a recombination event happens, it may become a recombination event if the Holliday junction is resolved/repared successfully, or a conversion event if the junction is not resolved/repared. The probability, however, is more commonly denoted by the ratio of conversion to recombination events in the literature. This ratio varies greatly from study to study, ranging from 0.1 to 15 (Chen et al, Nature Review Genetics, 2007). This translates to 0.1/0.9~0.1 to 15/16~0.94 of the gene conversion probability.

A *Recombinator* usually does not send any output. However, if an information field is given (parameter *infoFields*), this operator will treat this information field as a unique ID of parents and offspring and

output all recombination events in the format of `offspring_id parent_id starting_ploidy loc1 loc2 ... ``` where ``starting_ploidy indicates which homologous copy genotype replication starts from (0 or 1), `loc1`, `loc2` etc are loci after which recombination events happens. If there are multiple chromosomes on the genome, you will see a lot of (fake) recombination events because of independent segregation of chromosomes. Such a record will be generated for each set of homologous chromosomes so an diploid offspring will have two lines of output. Note that individual IDs need to be set (using a *IdTagger* operator) before this Recombinator is applied.

In addition to genotypes, this operator also copies allelic lineage if it is executed in a module with lineage allele type.

Note: There is no recombination between sex chromosomes (Chromosomes X and Y), although recombination is possible between pseudoautosomal regions on these chromosomes. If such a feature is required, you will have to simulate the pseudoautosomal regions as separate chromosomes.

transmitGenotype (*parent*, *offspring*, *ploidy*)

This function transmits genotypes from a *parent* to the *ploidy*-th homologous set of chromosomes of an *offspring*. It can be used, for example, by a customized genotype transmitter to use sex-specific recombination rates to transmit parental genotypes to offspring.

11.6 Mutation

11.6.1 class BaseMutator

class BaseMutator

Class `mutator` is the base class of all mutators. It handles all the work of picking an allele at specified loci from certain (virtual) subpopulation with certain probability, and calling a derived mutator to mutate the allele. Alleles can be changed before and after mutation if existing allele numbers do not match those of a mutation model.

BaseMutator (*rates*=[], *loci*=ALL_AVAIL, *mapIn*=[], *mapOut*=[], *context*=0, *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

A mutator mutates alleles from one state to another with given probability. This base mutator does not perform any mutation but it defines common behaviors of all mutators.

By default, a mutator mutates all alleles in all populations of a simulator at all generations. A number of parameters can be used to restrict mutations to certain generations (parameters *begin*, *end*, *step* and *at*), replicate populations (parameter *rep*), (virtual) subpopulations (parameter *subPops*) and loci (parameter *loci*). Parameter *loci* can be a list of loci indexes, names, list of chromosome position pairs, ALL_AVAIL, or a function with optional parameter *pop* that will be called at each generation to determine indexes of loci. Please refer to class *BaseOperator* for a detailed explanation of these parameters.

Parameter *rate* or its equivalence specifies the probability that a mutation event happens. The exact form and meaning of *rate* is mutator-specific. If a single rate is specified, it will be applied to all *loci*. If a list of mutation rates are given, they will be applied to each locus specified in parameter *loci*. Note that not all mutators allow specification of multiple mutation rate, especially when the mutation rate itself is a list or matrix.

Alleles at a locus are non-negative numbers 0, 1, ... up to the maximum allowed allele for the loaded module (1 for binary, 255 for short and 65535 for long modules). Whereas some general mutation models treat alleles as numbers, other models assume specific interpretation of alleles. For example, an *AcgtMutator* assumes alleles 0, 1, 2 and 3 as nucleotides A, C, G and T. Using a mutator that is incompatible with your simulation will certainly yield erroneous results.

If your simulation assumes different alleles with a mutation model, you can map an allele to the allele used in the model and map the mutated allele back. This is achieved using a *mapIn* list with its *i*-th item being the corresponding allele of real allele *i*, and a *mapOut* list with its *i*-th item being the real allele of allele *i* assumed in the model. For example *mapIn*=[0, 0, 1] and *mapOut*=[1, 2] would allow the use of a *SNPMutator* to mutate between alleles 1 and 2, instead of 0 and 1. Parameters *mapIn* and *mapOut* also accept a user-defined Python function that returns a corresponding allele for a given allele. This allows easier mapping between a large number of alleles and advanced models such as random emission of alleles.

If a valid information field is specified for parameter *infoFields* (default to *ind_id*) for modules with lineage allele type, the lineage of the mutated alleles will be the ID (stored in the first field of *infoFields*) of individuals that harbor the mutated alleles if *lineageMode* is set to *FROM_INFO* (default). If *lineageMode* is set to *FROM_INFO_SIGNED*, the IDs will be assigned a sign depending on the ploidy the mutation happens (1 for ploidy 0, -1 for ploidy 1, etc). The lineage information will be transmitted along with the alleles so this feature allows you to track the source of mutants during evolution.

A mutator by default does not produce any output. However, if a non-empty output is specified, the operator will output generation number, locus, ploidy, original allele, mutant, and values of all information field specified by parameter *infoFields* (e.g. individual ID if *ind_id* is specified).

Some mutation models are context dependent. Namely, how an allele mutates will depend on its adjacent alleles. Whereas most simuPOP mutators are context independent, some of them accept a parameter *context* which is the number of alleles to the left and right of the mutated allele. For example *context=1* will make the alleles to the immediate left and right to a mutated allele available to a mutator. These alleles will be mapped in if parameter *mapIn* is defined. How exactly a mutator makes use of these information is mutator dependent.

11.6.2 class MatrixMutator

class MatrixMutator

A matrix mutator mutates alleles 0, 1, ..., *n*-1 using a *n* by *n* matrix, which specifies the probability at which each allele mutates to another. Conceptually speaking, this mutator goes through all mutable allele and mutate it to another state according to probabilities in the corresponding row of the rate matrix. Only one mutation rate matrix can be specified which will be used for all specified loci. #

MatrixMutator (*rate*, *loci*=ALL_AVAIL, *mapIn*=[], *mapOut*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

Create a mutator that mutates alleles 0, 1, ..., *n*-1 using a *n* by *n* matrix *rate*. Item (*i*, *j*) of this matrix specifies the probability at which allele *i* mutates to allele *j*. Diagonal items (*i*, *i*) are ignored because they are automatically determined by other probabilities. Only one mutation rate matrix can be specified which will be used for all loci in the applied population, or loci specified by parameter *loci*. If alleles other than 0, 1, ..., *n*-1 exist in the population, they will not be mutated although a warning message will be given if debugging code *DBG_WARNING* is turned on. Please refer to classes *mutator* and *BaseOperator* for detailed explanation of other parameters.

11.6.3 class KAlleleMutator

class KAlleleMutator

This mutator implements a *k-allele* mutation model that assumes *k* allelic states (alleles 0, 1, 2, ..., *k*-1) at each locus. When a mutation event happens, it mutates an allele to any other states with equal probability.

KAlleleMutator (*k*, *rates*=[], *loci*=ALL_AVAIL, *mapIn*=[], *mapOut*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

Create a *k*-allele mutator that mutates alleles to one of the other *k*-1 alleles with equal probability. This

mutator by default applies to all loci unless parameter *loci* is specified. A single mutation rate will be used for all loci if a single value of parameter *rates* is given. Otherwise, a list of mutation rates can be specified for each locus in parameter *loci*. If the mutated allele is larger than or equal to *k*, it will not be mutated. A warning message will be displayed if debugging code `DBG_WARNING` is turned on. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

11.6.4 class StepwiseMutator

class StepwiseMutator

A stepwise mutation model treats alleles at a locus as the number of tandem repeats of microsatellite or minisatellite markers. When a mutation event happens, the number of repeats (allele) either increase or decrease. A standard stepwise mutation model increases or decreases an allele by 1 with equal probability. More complex models (generalized stepwise mutation model) are also allowed. Note that an allele cannot be mutated beyond boundaries (0 and maximum allowed allele).

StepwiseMutator (*rates*=[], *loci*=ALL_AVAIL, *incProb*=0.5, *maxAllele*=0, *mutStep*=[], *mapIn*=[], *mapOut*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

Create a stepwise mutation mutator that mutates an allele by increasing or decreasing it. This mutator by default applies to all loci unless parameter *loci* is specified. A single mutation rate will be used for all loci if a single value of parameter *rates* is given. Otherwise, a list of mutation rates can be specified for each locus in parameter *loci*.

When a mutation event happens, this operator increases or decreases an allele by *mutStep* steps. Acceptable input of parameter *mutStep* include

- A number: This is the default mode with default value 1.
- (GEOMETRIC_DISTRIBUTION, *p*): The number of steps follows a geometric distribution with parameter *p*.
- A Python function: This user defined function accepts the allele being mutated and return the steps to mutate.

The mutation process is usually neutral in the sense that mutating up and down is equally likely. You can adjust parameter *incProb* to change this behavior.

If you need to use other generalized stepwise mutation models, you can implement them using a `PyMutator`. If performance becomes a concern, I may add them to this operator if provided with a reliable reference.

11.6.5 class PyMutator

class PyMutator

This hybrid mutator accepts a Python function that determines how to mutate an allele when an mutation event happens.

PyMutator (*rates*=[], *loci*=ALL_AVAIL, *func*=None, *context*=0, *mapIn*=[], *mapOut*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

Create a hybrid mutator that uses a user-provided function to mutate an allele when a mutation event happens. This function (parameter *func*) accepts the allele to be mutated as parameter *allele*, locus index *locus*, and optional array of alleles as parameter *context*, which are *context* alleles the left and right of the mutated allele. Invalid context alleles (e.g. left allele to the first locus of a chromosome) will be marked by -1. The return value of this function will be used to mutate the passed allele. The passed, returned and context alleles might be altered if parameter *mapIn* and *mapOut* are used. This mutator by default applies to all loci unless parameter *loci* is specified. A single mutation rate will be used for all loci

if a single value of parameter *rates* is given. Otherwise, a list of mutation rates can be specified for each locus in parameter *loci*. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

11.6.6 class `MixedMutator`

class `MixedMutator`

This mixed mutator accepts a list of mutators and use one of them to mutate an allele when an mutation event happens.

MixedMutator (*rates*=[], *loci*=ALL_AVAIL, *mutators*=[], *prob*=[], *mapIn*=[], *mapOut*=[], *context*=0, *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

Create a mutator that randomly chooses one of the specified *mutators* to mutate an allele when a mutation event happens. The mutators are chosen according to a list of probabilities (parameter *prob*) that should add up to 1. The passed and returned alleles might be changed if parameters *mapIn* and *mapOut* are used. Most parameters, including *loci*, *mapIn*, *mapOut*, *rep*, and *subPops* of mutators specified in parameter *mutators* are ignored. This mutator by default applies to all loci unless parameter *loci* is specified. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

11.6.7 class `ContextMutator`

class `ContextMutator`

This context-dependent mutator accepts a list of mutators and use one of them to mutate an allele depending on the context of the mutated allele.

ContextMutator (*rates*=[], *loci*=ALL_AVAIL, *mutators*=[], *contexts*=[], *mapIn*=[], *mapOut*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

Create a mutator that choose one of the specified *mutators* to mutate an allele when a mutation event happens. The mutators are chosen according to the context of the mutated allele, which is specified as a list of alleles to the left and right of an allele (parameter *contexts*). For example, *contexts*=[(0, 0) , (0, 1) , (1, 1)] indicates which mutators should be used to mutate allele X in the context of 0X0, 0X1, and 1X1. A context can include more than one alleles at both left and right sides of a mutated allele but all contexts should have the same (even) number of alleles. If an allele does not have full context (e.g. when a locus is the first locus on a chromosome), unavailable alleles will be marked as -1. There should be a mutator for each context but an additional mutator can be specified as the default mutator for unmatched contexts. If parameters *mapIn* is specified, both mutated allele and its context alleles will be mapped. Most parameters, including *loci*, *mapIn*, *mapOut*, *rep*, and *subPops* of mutators specified in parameter *mutators* are ignored. This mutator by default applies to all loci unless parameter *loci* is specified. Please refer to classes `mutator` and `BaseOperator` for descriptions of other parameters.

11.6.8 class `PointMutator`

class `PointMutator`

A point mutator is different from all other mutators because mutations in this mutator do not happen randomly. Instead, it happens to specific loci and mutate an allele to a specific state, regardless of its original state. This mutator is usually used to introduce a mutant to a population.

PointMutator (*loci*, *allele*, *ploidy*=0, *inds*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=0, *infoFields*=["ind_id"], *lineageMode*=FROM_INFO)

Create a point mutator that mutates alleles at specified *loci* to a given *allele* of individuals *inds*. If there are multiple alleles at a locus (e.g. individuals in a diploid population), only the first allele is mutated unless

indexes of alleles are listed in parameter *ploidy*. This operator is by default applied to individuals in the first subpopulation but you can apply it to a different or more than one (virtual) subpopulations using parameter *subPops* (*AllAvail* is also accepted). Please refer to class *BaseOperator* for detailed descriptions of other parameters.

11.6.9 class SNPMutator

class SNPMutator

A mutator model that assumes two alleles 0 and 1 and accepts mutation rate from 0 to 1, and from 1 to 0 alleles.

SNPMutator (*u=0, v=0, loci=True, mapIn=[], mapOut=[], output="", begin=0, end=-1, step=1, at=[], reps=True, subPops=ALL_AVAIL, infoFields=['ind_id'], lineageMode=115*)

Return a *MatrixMutator* with proper mutate matrix for a two-allele mutation model using mutation rate from allele 0 to 1 (parameter *u*) and from 1 to 0 (parameter *v*)

11.6.10 class AcgtMutator

class AcgtMutator

This mutation operator assumes alleles 0, 1, 2, 3 as nucleotides A, C, G and T and use a 4 by 4 mutation rate matrix to mutate them. Although a general model needs 12 parameters, less parameters are needed for specific nucleotide mutation models (parameter *model*). The length and meaning of parameter *rate* is model dependent.

AcgtMutator (*rate=[], model='general', loci=True, mapIn=[], mapOut=[], output="", begin=0, end=-1, step=1, at=[], reps=True, subPops=ALL_AVAIL, infoFields=['ind_id'], lineageMode=115*)

Create a mutation model that mutates between nucleotides A, C, G, and T (alleles are coded in that order as 0, 1, 2 and 3). Currently supported models are Jukes and Cantor 1969 model (JC69), Kimura's 2-parameter model (K80), Felsenstein 1981 model (F81), Hasgawa, Kishino and Yano 1985 model (HKY85), Tamura 1992 model (T92), Tamura and Nei 1993 model (TN93), Generalized time reversible model (GTR), and a general model (*general*) with 12 parameters. Please refer to the simuPOP user's guide for detailed information about each model.

11.7 Penetrance

11.7.1 class BasePenetrance

class BasePenetrance

A penetrance model models the probability that an individual has a certain disease provided that he or she has certain genetic (genotype) and environmental (information field) risk factors. A penetrance operator calculates this probability according to provided information and set his or her affection status randomly. For example, an individual will have probability 0.8 to be affected if the penetrance is 0.8. This class is the base class to all penetrance operators and defines a common interface for all penetrance operators.

A penetrance operator can be applied at any stage of an evolutionary cycle. If it is applied before or after mating, it will set affection status of all parents and offspring, respectively. If it is applied during mating, it will set the affection status of each offspring. You can also apply a penetrance operator to an individual using its *applyToIndividual* member function.

By default, a penetrance operator assigns affection status of individuals but does not save the actual penetrance value. However, if an information field is specified, penetrance values will be saved to this field for future analysis.

When a penetrance operator is applied to a population, it is only applied to the current generation. You can, however, use parameter *ancGens* to set affection status for all ancestral generations (`ALL_AVAIL`), or individuals in specified generations if a list of ancestral generations is specified. Note that this parameter is ignored if the operator is applied during mating.

BasePenetrance (*ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a base penetrance operator. This operator assign individual affection status in the present generation (default). If `ALL_AVAIL` or a list of ancestral generations are specified in parameter *ancGens*, individuals in specified ancestral generations will be processed. A penetrance operator can be applied to specified (virtual) subpopulations (parameter *subPops*) and replicates (parameter *reps*). If an informatio field is given, penetrance value will be stored in this information field of each individual.

apply (*pop*)

set penetrance to all individuals and record penetrance if requested

applyToIndividual (*ind, pop=None*)

Apply the penetrance operator to a single individual *ind* and set his or her affection status. A population reference can be passed if the penetrance model depends on population properties such as generation number. This function returns the affection status.

11.7.2 class MapPenetrance

class MapPenetrance

This penetrance operator assigns individual affection status using a user-specified penetrance dictionary.

MapPenetrance (*loci, penetrance, ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a penetrance operator that get penetrance value from a dictionary *penetrance* with genotype at *loci* as keys, and *penetrance* as values. For each individual, genotypes at *loci* are collected one by one (e.g. `p0_loc0, p1_loc0, p0_loc1, p1_loc1...` for a diploid individual) and are looked up in the dictionary. Parameter *loci* can be a list of loci indexes, names, list of chromosome position pairs, `ALL_AVAIL`, or a function with optional parameter `pop` that will be called at each ganeeration to determine indexes of loci. If a genotype cannot be found, it will be looked up again without phase information (e.g. `(1, 0)` will match key `(0, 1)`). If the genotype still can not be found, a `ValueError` will be raised. This operator supports sex chromosomes and haplodiploid populations. In these cases, only valid genotypes should be used to generator the dictionary keys.

11.7.3 class MaPenetrance

class MaPenetrance

This operator is called a ‘multi-allele’ penetrance operator because it groups multiple alleles into two groups: wildtype and non-wildtype alleles. Alleles in each allele group are assumed to have the same effect on individual penetrance. If we denote all wildtype alleles as *A*, and all non-wildtype alleles *a*, this operator assign Individual penetrance according to genotype *AA*, *Aa*, *aa* in the diploid case, and *A* and *a* in the haploid case.

MaPenetrance (*loci, penetrance, wildtype=0, ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Creates a multi-allele penetrance operator that groups multiple alleles into a wildtype group (with alleles *wildtype*, default to `[0]`), and a non-wildtype group. A list of penetrance values is specified through parameter *penetrance*, for genotypes at one or more *loci*. Parameter *loci* can be a list of loci indexes, names, list of chromosome position pairs, `ALL_AVAIL`, or a function with optional parameter `pop` that will be called at each ganeeration to determine indexes of loci. If we denote wildtype alleles using capital letters *A*, *B* ... and non-wildtype alleles using small letters *a*, *b* ..., the penetrance values should be for

- genotypes *A* and *a* for the haploid single-locus case,

- genotypes AB, Ab, aB and bb for haploid two=locus cases,
- genotypes AA, Aa and aa for diploid single-locus cases,
- genotypes AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, and aabb for diploid two- locus cases,
- and in general 2^n for diploid and 3^n for haploid cases if there are n loci.

This operator does not support haplodiploid populations and sex chromosomes.

11.7.4 class MIPenetrance

class MIPenetrance

This penetrance operator is created by a list of penetrance operators. When it is applied to an individual, it applies these penetrance operators to the individual, obtain a list of penetrance values, and compute a combined penetrance value from them and assign affection status accordingly. ADDITIVE, multiplicative, and a heterogeneous multi-locus model are supported. Please refer to Neil Rish (1989) “Linkage Strategies for Genetically Complex Traits” for some analysis of these models.

MIPenetrance (*ops, mode=MULTIPLICATIVE, ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a multiple-locus penetrance operator from a list penetrance operator *ops*. When this operator is applied to an individual (parents when used before mating and offspring when used during mating), it applies these operators to the individual and obtain a list of (usually single-locus) penetrance values. These penetrance values are combined to a single penetrance value using

- $Prod(f_i)$, namely the product of individual penetrance if *mode* = MULTIPLICATIVE,
- $sum(f_i)$ if *mode* = ADDITIVE, and
- $1-Prod(1 - f_i)$ if *mode* = HETEROGENEITY

0 or 1 will be returned if the combined penetrance value is less than zero or greater than 1.

Applicability parameters (begin, end, step, at, reps, subPops) could be used in both *MISelector* and selectors in parameter *ops*, but parameters in *MISelector* will be interpreted first.

11.7.5 class PyPenetrance

class PyPenetrance

This penetrance operator assigns penetrance values by calling a user provided function. It accepts a list of loci (parameter *loci*), and a Python function *func* which should be defined with one or more of parameters *geno*, *mut*, *gen*, *ind*, *pop*, or names of information fields. When this operator is applied to a population, it passes genotypes or mutants (non-zero alleles) at specified loci at specified loci, generation number, a reference to an individual, a reference to the current population (usually used to retrieve population variables) and values at specified information fields to respective parameters of this function. Genotypes of each individual are passed as a tuple of alleles arranged locus by locus (in the order of A1,A2,B1,B2 for loci A and B). Mutants are passed as a default dictionary of loci index (with respect to all genotype of individuals, not just the first ploidy) and alleles. The returned penetrance values will be used to determine the affection status of each individual.

PyPenetrance (*func, loci=[], ancGens=UNSPECIFIED, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a Python hybrid penetrance operator that passes genotype at specified *loci*, values at specified information fields (if requested), and a generation number to a user-defined function *func*. Parameter *loci* can be a list of loci indexes, names, list of chromosome position pairs, ALL_AVAIL, or a function with optional parameter *pop* that will be called at each generation to determine indexes of loci. The return value will be treated as Individual penetrance.

11.7.6 class PyMIPenetrance

class PyMIPenetrance

This penetrance operator is a multi-locus Python penetrance operator that assigns penetrance values by combining locus and genotype specific penetrance values. It differs from a *PyPenetrance* in that the python function is responsible for penetrance values for each genotype type at each locus, which can potentially be random, and locus or genotype-specific.

PyMIPenetrance (*func*, *mode*=MULTIPLICATIVE, *loci*=ALL_AVAIL, *ancGens*=UNSPECIFIED, *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=[])

Create a penetrance operator that assigns individual affection status according to penetrance values combined from locus- specific penetrance values that are determined by a Python call- back function. The callback function accepts parameter *loc*, *alleles* (both optional) and returns location- or genotype- specific penetrance values that can be constant or random. The penetrance values for each genotype will be cached so the same penetrance values will be assigned to genotypes with previously assigned values. Note that a function that does not examine the genotype naturally assumes a dominant model where genotypes with one or two mutants have the same penetrance value. Because genotypes at a locus are passed separately and in no particular order, this function is also responsible for assigning consistent fitness values for genotypes at the same locus (a class is usually used). This operator currently ignores chromosome types so unused alleles will be passed for loci on sex or mitochondrial chromosomes. This operator also ignores the phase of genotype so genotypes (a,b) and (b,a) are assumed to have the same fitness effect.

Individual penetrance will be combined in ADDITIVE, MULTIPLICATIVE, or HETEROGENEITY mode from penetrance values of loci with at least one non-zero allele (See *MIPenetrance* for details).

11.8 Quantitative Trait

11.8.1 class BaseQuanTrait

class BaseQuanTrait

A quantitative trait in simuPOP is simply an information field. A quantitative trait model simply assigns values to one or more information fields (called trait fields) of each individual according to its genetic (genotype) and environmental (information field) factors. It can be applied at any stage of an evolutionary cycle. If a quantitative trait operator is applied before or after mating, it will set the trait fields of all parents and offspring. If it is applied during mating, it will set the trait fields of each offspring.

When a quantitative trait operator is applied to a population, it is only applied to the current generation. You can, however, use parameter *ancGen*=-1 to set the trait field of all ancestral generations, or a generation index to apply to only ancestral generation younger than *ancGen*. Note that this parameter is ignored if the operator is applied during mating.

BaseQuanTrait (*ancGens*=UNSPECIFIED, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=[])

Create a base quantitative trait operator. This operator assigns one or more quantitative traits to trait fields in the present generation (default). If ALL_AVAIL or a list of ancestral generations are specified, this operator will be applied to individuals in these generations as well. A quantitative trait operator can be applied to specified (virtual) subpopulations (parameter *subPops*) and replicates (parameter *reps*).

apply (*pop*)

set *qtrait* to all individual

11.8.2 class PyQuanTrait

class PyQuanTrait

This quantitative trait operator assigns a trait field by calling a user provided function. It accepts a list of loci (parameter *loci*), and a Python function *func* which should be defined with one or more of parameters *geno*, *mut*, *gen*, *ind*, or names of information fields. When this operator is applied to a population, it passes genotypes or mutants (non-zero alleles) of each individual at specified loci, generation number, a reference to an individual, and values at specified information fields to respective parameters of this function. Genotypes of each individual are passed as a tuple of alleles arranged locus by locus (in the order of A1,A2,B1,B2 for loci A and B). Mutants are passed as a default dictionary of loci index (with respect to all genotype of individuals, not just the first ploidy) and alleles. The return values will be assigned to specified trait fields.

PyQuanTrait (*func*, *loci*=[], *ancGens*=UNSPECIFIED, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=[])

Create a Python hybrid quantitative trait operator that passes genotype at specified *loci*, optional values at specified information fields (if requested), and an optional generation number to a user-defined function *func*. Parameter *loci* can be a list of loci indexes, names, or ALL_AVAIL. The return value will be assigned to specified trait fields (*infoField*). If only one trait field is specified, a number or a sequence of one element is acceptable. Otherwise, a sequence of values will be accepted and be assigned to each trait field.

11.9 Natural selection

11.9.1 class BaseSelector

class BaseSelector

This class is the base class to all selectors, namely operators that perform natural selection. It defines a common interface for all selectors.

A selector can be applied before mating or during mating. If a selector is applied to one or more (virtual) subpopulations of a parental population before mating, it sets individual fitness values to all involved parents to an information field (default to *fitness*). When a mating scheme that supports natural selection is applied to the parental population, it will select parents with probabilities that are proportional to individual fitness stored in an information field (default to *fitness*). Individual fitness is considered **relative** fitness and can be any non-negative number. This simple process has some implications that can lead to advanced usages of natural selection in simuPOP:

- It is up to the mating scheme how to handle individual fitness. Some mating schemes do not support natural selection at all.
- A mating scheme performs natural selection according to fitness values stored in an information field. It does not care how these values are set. For example, fitness values can be inherited from a parent using a tagging operator, or set directly using a Python operator.
- A mating scheme can treat any information field as fitness field. If an specified information field does not exist, or if all individuals have the same fitness values (e.g. 0), the mating scheme selects parents randomly.
- Multiple selectors can be applied to the same parental generation. individual fitness is determined by the last fitness value it is assigned.
- A selection operator can be applied to virtual subpopulations and set fitness values only to part of the individuals.
- individuals with zero fitness in a subpopulation with anyone having a positive fitness value will not be selected to produce offspring. This can sometimes lead to unexpected behaviors. For example, if you only assign fitness value to part of the individuals in a subpopulation, the rest of them will be effectively

discarded. If you migrate individuals with valid fitness values to a subpopulation with all individuals having zero fitness, the migrants will be the only mating parents.

- It is possible to assign multiple fitness values to different information fields so that different homogeneous mating schemes can react to different fitness schemes when they are used in a heterogeneous mating scheme.
- You can apply a selector to the offspring generation using the *postOps* parameter of *Simulator.evolve*, these fitness values will be used when the offspring generation becomes parental generation in the next generation.

Alternatively, a selector can be used as a during mating operator. In this case, it calculates fitness value for each offspring which will be treated as **absolute** fitness, namely the probability for each offspring to survive. This process uses the fact that an individual will be discarded when any of the during mating operators returns *False*. It is important to remember that:

- individual fitness needs to be between 0 and 1 in this case.
- Fitness values are not stored so the population does not need an information field *fitness*.
- This method applies natural selection to offspring instead of parents. These two implementation can be identical or different depending on the mating scheme used.
- Selecting offspring is less efficient than the selecting parents, especially when fitness values are low.
- Parameter *subPops* are applied to the offspring population and is used to judge if an operator should be applied. It thus does not make sense to apply a selector to a virtual subpopulation with affected individuals.

BaseSelector (*output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=ALL_AVAIL*)

Create a base selector object. This operator should not be created directly.

11.9.2 class MapSelector

class MapSelector

This selector assigns individual fitness values using a user- specified dictionary. This operator can be applied to populations with arbitrary number of homologous chromosomes.

MapSelector (*loci*, *fitness*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=ALL_AVAIL*)

Create a selector that assigns individual fitness values using a dictionary *fitness* with genotype at *loci* as keys, and fitness as values. Parameter *loci* can be a list of indexes, loci names, list of chromosome position pairs, *ALL_AVAIL*, or a function with optional parameter *pop* that will be called at each generation to determine indexes of loci. For each individual (parents if this operator is applied before mating, and offspring if this operator is applied during mating), genotypes at *loci* are collected one by one (e.g. *p0_loc0*, *p1_loc0*, *p0_loc1*, *p1_loc1*... for a diploid individual, with number of alleles varying for sex and mitochondrial DNAs) and are looked up in the dictionary. If a genotype cannot be found, it will be looked up again without phase information (e.g. *(1, 0)* will match key *(0, 1)*). If the genotype still can not be found, a *ValueError* will be raised. This operator supports sex chromosomes and haplodiploid populations. In these cases, only valid genotypes should be used to generator the dictionary keys.

11.9.3 class MaSelector

class MaSelector

This operator is called a ‘multi-allele’ selector because it groups multiple alleles into two groups: wildtype and non-wildtype alleles. Alleles in each allele group are assumed to have the same effect on individual fitness. If we denote all wildtype alleles as *A*, and all non-wildtype alleles *a*, this operator assign individual fitness according to genotype *AA*, *Aa*, *aa* in the diploid case, and *A* and *a* in the haploid case.

MaSelector (*loci*, *fitness*, *wildtype*=0, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=ALL_AVAIL)

Creates a multi-allele selector that groups multiple alleles into a wildtype group (with alleles *wildtype*, default to [0]), and a non-wildtype group. A list of fitness values is specified through parameter *fitness*, for genotypes at one or more *loci*. Parameter *loci* can be a list of indexes, loci names, list of chromosome position pairs, ALL_AVAIL, or a function with optional parameter *pop* that will be called at each generation to determine indexes of loci. If we denote wildtype alleles using capital letters A, B ... and non-wildtype alleles using small letters a, b ..., the fitness values should be for

- genotypes A and a for the haploid single-locus case,
- genotypes AB, Ab, aB and bb for haploid two-locus cases,
- genotypes AA, Aa and aa for diploid single-locus cases,
- genotypes AABB, AABb, AAbb, AaBB, AaBb, Aabb, aaBB, aaBb, and aabb for diploid two-locus cases,
- and in general 2^n for diploid and 3^n for haploid cases if there are *n* loci.

This operator does not support haplodiploid populations, sex and mitochondrial chromosomes.

11.9.4 class MlSelector

class MlSelector

This selector is created by a list of selectors. When it is applied to an individual, it applies these selectors to the individual, obtain a list of fitness values, and compute a combined fitness value from them. ADDITIVE, multiplicative, and a heterogeneous multi-locus model are supported.

MlSelector (*ops*, *mode*=MULTIPLICATIVE, *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=ALL_AVAIL)

Create a multiple-locus selector from a list selection operator *selectors*. When this operator is applied to an individual (parents when used before mating and offspring when used during mating), it applies these operators to the individual and obtain a list of (usually single-locus) fitness values. These fitness values are combined to a single fitness value using

- $Prod(f_i)$, namely the product of individual fitness if *mode* = MULTIPLICATIVE,
- $1 - sum(1 - f_i)$ if *mode* = ADDITIVE,
- $1 - Prod(1 - f_i)$ if *mode* = HETEROGENEITY, and
- $exp(-sum(1 - f_i))$ if *mode* = EXPONENTIAL,

zero will be returned if the combined fitness value is less than zero.

Applicability parameters (*begin*, *end*, *step*, *at*, *reps*, *subPops*) could be used in both *MlSelector* and selectors in parameter *ops*, but parameters in *MlSelector* will be interpreted first.

11.9.5 class PySelector

class PySelector

This selector assigns fitness values by calling a user provided function. It accepts a list of loci (parameter *loci*) and a Python function *func* which should be defined with one or more of parameters *geno*, *mut*, *gen*, *ind*, *pop* or names of information fields. Parameter *loci* can be a list of loci indexes, names, list of chromosome position pairs, ALL_AVAIL, or a function with optional parameter *pop* that will be called at each generation to determine indexes of loci. When this operator is applied to a population, it passes genotypes or mutants at specified loci, generation number, a reference to an individual, a reference to the current population (usually used to retrieve population variable), and values at specified information fields to respective parameters of this

function. Genotypes are passed as a tuple of alleles arranged locus by locus (in the order of A1,A2,B1,B2 for loci A and B). Mutants are passed as a default dictionary of loci index (with respect to all genotype of individuals, not just the first ploidy) and alleles. The returned value will be used to determine the fitness of each individual.

PySelector (*func*, *loci*=[], *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *output*="", *subPops*=ALL_AVAIL, *infoFields*=ALL_AVAIL)

Create a Python hybrid selector that passes genotype at specified *loci*, values at specified information fields (if requested) and a generation number to a user-defined function *func*. The return value will be treated as individual fitness.

11.9.6 class PyMlSelector

class PyMlSelector

This selector is a multi-locus Python selector that assigns fitness to individuals by combining locus and genotype specific fitness values. It differs from a *PySelector* in that the python function is responsible for assigning fitness values for each genotype type at each locus, which can potentially be random, and locus or genotype-specific.

PyMlSelector (*func*, *mode*=EXPONENTIAL, *loci*=ALL_AVAIL, *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=ALL_AVAIL, *subPops*=ALL_AVAIL, *infoFields*=ALL_AVAIL)

Create a selector that assigns individual fitness values by combining locus-specific fitness values that are determined by a Python call-back function. The callback function accepts parameter *loc*, *alleles* (both optional) and returns location- or genotype-specific fitness values that can be constant or random. The fitness values for each genotype will be cached so the same fitness values will be assigned to genotypes with previously assigned values. Note that a function that does not examine the genotype naturally assumes a dominant model where genotypes with one or two mutants have the same fitness effect. Because genotypes at a locus are passed separately and in no particular order, this function is also responsible for assigning consistent fitness values for genotypes at the same locus (a class is usually used). This operator currently ignores chromosome types so unused alleles will be passed for loci on sex or mitochondrial chromosomes. It also ignores phase of genotype so it will use the same fitness value for genotype (a,b) and (b,a).

Individual fitness will be combined in ADDITIVE, MULTIPLICATIVE, HETEROGENEITY, or EXPONENTIAL mode from fitness values of loci with at least one non-zero allele (See *MlSelector* for details). If an output is given, location, genotype, fitness and generation at which the new genotype is assigned the value will be written to the output, in the format of 'loc a1 a2 fitness gen' for loci on autosomes of diploid populations.

11.10 Tagging operators

11.10.1 class IdTagger

class IdTagger

An *IdTagger* gives a unique ID for each individual it is applies to. These ID can be used to uniquely identify an individual in a multi-generational population and be used to reliably reconstruct a Pedigree.

To ensure uniqueness across populations, a single source of ID is used for this operator. individual IDs are assigned consecutively starting from 1. Value 1 instead of 0 is used because most software applications use 0 as missing values for parentship. If you would like to reset the sequence or start from a different number, you can call the *reset* (*startID*) function of any *IdTagger*.

An *IdTagger* is usually used during-mating to assign ID to each offspring. However, if it is applied directly to a population, it will assign unique IDs to all individuals in this population. This property is usually used in the *preOps* parameter of function *Simulator.evolve* to assign initial ID to a population.

IdTagger (*begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=["ind_id"]*)

Create an *IdTagger* that assign an unique ID for each individual it is applied to. The IDs are created sequentially and are stored in an information field specified in parameter *infoFields* (default to *ind_id*). This operator is considered a during-mating operator but it can be used to set ID for all individuals of a population when it is directly applied to the population.

reset (*startID=1*)

Reset the global individual ID number so that *IdTaggers* will start from *id* (default to 1) again.

11.10.2 class InheritTagger

class InheritTagger

An inheritance tagger passes values of parental information field(s) to the corresponding fields of offspring. If there are two parental values from parents of a sexual mating event, a parameter *mode* is used to specify how to assign offspring information fields.

InheritTagger (*mode=PATERNAL, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=[]*)

Creates an inheritance tagger that passes values of parental information fields (parameter *infoFields*) to the corresponding fields of offspring. If there is only one parent, values at the specified information fields are copied directly. If there are two parents, parameter *mode* specifies how to pass them to an offspring. More specifically,

- *mode*=MATERNAL Passing the value from mother.
- *mode*=PATERNAL Passing the value from father.
- *mode*=MEAN Passing the average of two values.
- *mode*=MAXIMUM Passing the maximum value of two values.
- *mode*=MINIMUM Passing the minimum value of two values.
- *mode*=SUMMATION Passing the summation of two values.
- *mode*=MULTIPLICATION Passing the multiplication of two values.

An *RuntimeError* will be raised if any of the parents does not exist. This operator does not support parameter *subPops* and does not output any information.

11.10.3 class SummaryTagger

class SummaryTagger

A summary tagger summarize values of one or more parental information field to another information field of an offspring. If mating is sexual, two sets of parental values will be involved.

SummaryTagger (*mode=MEAN, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=[]*)

Creates a summary tagger that summarize values of one or more parental information field (*infoFields*[:-1]*) to an offspring information field (**infoFields*[-1]*). A parameter **mode* specifies how to pass summarize parental values. More specifically,

- *mode*=MEAN Passing the average of values.
- *mode*=MAXIMUM Passing the maximum value of values.
- *mode*=Minumum Passing the minimum value of values.
- *mode*=SUMMATION Passing the sum of values.

- `mode=MULTIPLICATION` Passing the multiplication of values.

This operator does not support parameter *subPops* and does not output any information.

11.10.4 class ParentsTagger

class ParentsTagger

This tagging operator records the indexes of parents (relative to the parental generation) of each offspring in specified information fields (default to `father_idx` and `mother_idx`). Only one information field should be specified if an asexual mating scheme is used so there is one parent for each offspring. Information recorded by this operator is intended to be used to look up parents of each individual in multi-generational Population.

ParentsTagger (*begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output=""*, *infoFields=["father_idx", "mother_idx"]*)

Create a parents tagger that records the indexes of parents of each offspring when it is applied to an offspring during-mating. If two information fields are specified (parameter *infoFields*, with default value `['father_idx', 'mother_idx']`), they are used to record the indexes of each individual's father and mother. Value `-1` will be assigned if any of the parent is missing. If only one information field is given, it will be used to record the index of the first valid parent (father if both parents are valid). This operator ignores parameters *output* and *subPops*.

11.10.5 class OffspringTagger

class OffspringTagger

This tagging operator records the indexes of offspring within a family (sharing the same parent or parents) in specified information field (default to `offspring_idx`). This tagger can be used to control the number of offspring during mating.

OffspringTagger (*begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output=""*, *infoFields=ALL_AVAIL*)

Create an offspring tagger that records the indexes of offspring within a family. The index is determined by successful production of offspring during a mating events so the it does not increase the index if a previous offspring is discarded, and it resets index even if adjacent families share the same parents. This operator ignores parameters *stage*, *output*, and *subPops*.

11.10.6 class PedigreeTagger

class PedigreeTagger

This tagging operator records the ID of parents of each offspring in specified information fields (default to `father_id` and `mother_id`). Only one information field should be specified if an asexual mating scheme is used so there is one parent for each offspring. Information recorded by this operator is intended to be used to record full pedigree information of an evolutionary process.

PedigreeTagger (*idField="ind_id", output=""*, *outputFields=[], outputLocs=[], begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=["father_id", "mother_id"]*)

Create a pedigree tagger that records the ID of parents of each offspring when it is applied to an offspring during-mating. If two information fields are specified (parameter *infoFields*, with default value `['father_id', 'mother_id']`), they are used to record the ID of each individual's father and mother stored in the *idField* (default to `ind_id`) field of the parents. Value `-1` will be assigned if any of the parent is missing. If only one information field is given, it will be used to record the ID of the first valid parent (father if both pedigree are valid).

This operator by default does not send any output. If a valid output stream is given (should be in the form of `'>>filename'` so that output will be concatenated), this operator will output the ID of offspring,

IDs of his or her parent(s), sex and affection status of offspring, and values at specified information fields (*outputFields*) and loci (*outputLoci*) in the format of `off_id father_id mother_id M/F A/U fields genotype`. `father_id` or `mother_id` will be ignored if only one parent is involved. This file format can be loaded using function `loadPedigree`.

Because only offspring will be outputted, individuals in the top- most ancestral generation will not be outputted. This is usually not a problem because individuals who have offspring in the next generation will be constructed by function `loadPedigree`, although their information fields and genotype will be missing. If you would like to create a file with complete pedigree information, you can apply this operator before evolution in the *initOps* parameter of functions `Population.evolve` or `Simulator.evolve`. This will output all individuals in the initial population (the top-most ancestral population after evolution) in the same format. Note that sex, affection status and genotype can be changed by other operators so this operator should usually be applied after all other operators are applied.

11.10.7 class PyTagger

class PyTagger

A Python tagger takes some information fields from both parents, pass them to a user provided Python function and set the offspring individual fields with the return values.

PyTagger (*func=None, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, output="", infoFields=[]*)

Create a hybrid tagger that provides an user provided function *func* with values of specified information fields (determined by parameter names of this function) of parents and assign corresponding information fields of offspring with its return value. If more than one parent are available, maternal values are passed after paternal values. For example, if a function `func(A, B)` is passed, this operator will send two tuples with parental values of information fields 'A' and 'B' to this function and assign its return values to fields 'A' and 'B' of each offspring. The return value of this function should be a list, although a single value will be accepted if only one information field is specified. This operator ignores parameters *stage*, *output* and *subPops*.

11.11 Statistics Calculation

11.11.1 class Stat

class Stat

Operator *Stat* calculates various statistics of the population being applied and sets variables in its local namespace. Other operators or functions can retrieve results from or evaluate expressions in this local namespace after *Stat* is applied.

Stat (*popSize=False, numOfMales=False, numOfAffected=False, numOfSegSites=[], numOfMutants=[], alleleFreq=[], heteroFreq=[], homoFreq=[], genoFreq=[], haploFreq=[], haploHeteroFreq=[], haploHomoFreq=[], sumOfInfo=[], meanOfInfo=[], varOfInfo=[], maxOfInfo=[], minOfInfo=[], LD=[], association=[], neutrality=[], structure=[], HWE=[], inbreeding=[], effectiveSize=[], vars=ALL_AVAIL, suffix="", output="", begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Create a *Stat* operator that calculates specified statistics of a population when it is applied to this population. This operator can be applied to specified replicates (parameter *rep*) at specified generations (parameter *begin*, *end*, *step*, and *at*). This operator does not produce any output (ignore parameter *output*) after statistics are calculated. Instead, it stores results in the local namespace of the population being applied. Other operators can retrieve these variables or evaluate expression directly in this local namespace. Please refer to operator *BaseOperator* for a detailed explanation of these common operator parameters.

Stat supports parameter *subPops*. It usually calculate the same set of statistics for all subpopulations (*subPops*=*subPopList*()). If a list of (virtual) subpopulations are specified, statistics for only specified subpopulations will be calculated. However, different statistics treat this parameter differently and it is very important to check its reference before you use *subPops* for any statistics.

Calculated statistics are saved as variables in a population's local namespace. These variables can be numbers, lists or dictionaries and can be retrieved using functions `Population.vars()` or `Population.dvars()`. A special default dictionary (*defdict*) is used for dictionaries whose keys are determined dynamically. Accessing elements of such a dictionary with an invalid key will yield value 0 instead of a `KeyError`. If the same variables are calculated for one or more (virtual) subpopulation, the variables are stored in `vars()['subPop'][sp]['var']` where *sp* is a subpopulation ID (*sp*) or a tuple of virtual subpopulation ID ((*sp*, *vsp*)). `Population.vars(sp)` and `Population.dvars(sp)` provide shortcuts to these variables.

Operator *Stat* outputs a number of most useful variables for each type of statistic. For example, *alleleFreq* calculates both allele counts and allele frequencies and it by default sets variable *alleleFreq* (`dvars().alleleFreq`) for all or specified subpopulations. If this does not fit your need, you can use parameter *vars* to output additional parameters, or limit the output of existing parameters. More specifically, for this particular statistic, the available variables are 'alleleFreq', 'alleleNum', 'alleleFreq_sp' ('alleleFreq' in each subpopulation), and 'alleleNum_sp' ('alleleNum' in each subpopulation). You can set `vars=['alleleNum_sp']` to output only subpopulation specific allele count. An optional suffix (parameter *suffix*) can be used to append a suffix to default parameter names. This parameter can be used, for example, to calculate and store the same statistics for different subpopulations (e.g. pairwise *Fst*).

Operator *Stat* supports the following statistics:

popSize: If *popSize*=*True*, number of individuals in all or specified subpopulations (parameter *subPops*) will be set to the following variables:

- *popSize* (default): Number of individuals in all or specified subpopulations. Because *subPops* does not have to cover all individuals, it may not be the actual population size.
- *popSize_sp*: Size of (virtual) subpopulation *sp*.
- *subPopSize* (default): A list of (virtual) subpopulation sizes. This variable is easier to use than accessing *popSize* from each (virtual) subpopulation.

numOfMales: If *numOfMales*=*True*, number of male individuals in all or specified (virtual) subpopulations will be set to the following variables:

- *numOfMales* (default): Total number of male individuals in all or specified (virtual) subpopulations.
- *numOfFemales* (default): Total number of female individuals in all or specified (virtual) subpopulations.
- *propOfMales*: Proportion of male individuals.
- *propOfFemales*: Proportion of female individuals.
- *numOfMales_sp*: Number of male individuals in each (virtual) subpopulation.
- *numOfFemales_sp*: Number of female individuals in each (virtual) subpopulation.
- *propOfMales_sp*: Proportion of male individuals in each (virtual) subpopulation.
- *propOfFemales_sp*: Proportion of female individuals in each (virtual) subpopulation.

numOfAffected: If *numOfAffected*=*True*, number of affected individuals in all or specified (virtual) subpopulations will be set to the following variables:

- *numOfAffected* (default): Total number of affected individuals in all or specified (virtual) subpopulations.

- `numOfUnaffected` (default): Total number of unaffected individuals in all or specified (virtual) subpopulations.
- `propOfAffected`: Proportion of affected individuals.
- `propOfUnaffected`: Proportion of unaffected individuals.
- `numOfAffected_sp`: Number of affected individuals in each (virtual) subpopulation.
- `numOfUnaffected_sp`: Number of unaffected individuals in each (virtual) subpopulation.
- `propOfAffected_sp`: Proportion of affected individuals in each (virtual) subpopulation.
- `propOfUnaffected_sp`: Proportion of unaffected individuals in each (virtual) subpopulation.

numOfSegSites: Parameter *numOfSegSites* accepts a list of loci (loci indexes, names, or `ALL_AVAIL`) and count the number of loci with at least two different alleles (segregating sites) or loci with only one non-zero allele (no zero allele, not segregating) for individuals in all or specified (virtual) subpopulations. This parameter sets variables

- `numOfSegSites` (default): Number of segregating sites in all or specified (virtual) subpopulations.
- `numOfSegSites_sp`: Number of segregating sites in each (virtual) subpopulation.
- `numOfFixedSites`: Number of sites with one non-zero allele in all or specified (virtual) subpopulations.
- `numOfFixedSites_sp`: Number of sites with one non-zero allele in in each (virtual) subpopulations.
- `segSites`: A list of segregating sites in all or specified (virtual) subpopulations.
- `segSites_sp`: A list of segregating sites in each (virtual) subpopulation.
- `fixedSites`: A list of sites with one non-zero allele in all or specified (virtual) subpopulations.
- `fixedSites_sp`: A list of sites with one non-zero allele in in each (virtual) subpopulations.

numOfMutants: Parameter *numOfMutants* accepts a list of loci (loci indexes, names, or `ALL_AVAIL`) and count the number of mutants (non-zero alleles) for individuals in all or specified (virtual) subpopulations. It sets variables

- `numOfMutants` (default): Number of mutants in all or specified (virtual) subpopulations.
- `numOfMutants_sp`: Number of mutants in each (virtual) subpopulations.

alleleFreq: This parameter accepts a list of loci (loci indexes, names, or `ALL_AVAIL`), at which allele frequencies will be calculated. This statistic outputs the following variables, all of which are dictionary (with loci indexes as keys) of default dictionaries (with alleles as keys). For example, `alleleFreq[loc][a]` returns 0 if allele *a* does not exist.

- `alleleFreq` (default): `alleleFreq[loc][a]` is the frequency of allele *a* at locus *loc* for all or specified (virtual) subpopulations.
- `alleleNum` (default): `alleleNum[loc][a]` is the number of allele *a* at locus *loc* for all or specified (virtual) subpopulations.
- `alleleFreq_sp`: Allele frequency in each (virtual) subpopulation.
- `alleleNum_sp`: Allele count in each (virtual) subpopulation.

heteroFreq and **homoFreq:** These parameters accept a list of loci (by indexes or names), at which the number and frequency of homozygotes and/or heterozygotes will be calculated. These statistics are only available for diploid populations. The following variables will be outputted:

- `heteroFreq` (default for parameter *heteroFreq*): A dictionary of proportion of heterozygotes in all or specified (virtual) subpopulations, with loci indexes as dictionary keys.
- `homoFreq` (default for parameter *homoFreq*): A dictionary of proportion of homozygotes in all or specified (virtual) subpopulations.
- `heteroNum`: A dictionary of number of heterozygotes in all or specified (virtual) subpopulations.
- `homoNum`: A dictionary of number of homozygotes in all or specified (virtual) subpopulations.
- `heteroFreq_sp`: A dictionary of proportion of heterozygotes in each (virtual) subpopulation.
- `homoFreq_sp`: A dictionary of proportion of homozygotes in each (virtual) subpopulation.
- `heteroNum_sp`: A dictionary of number of heterozygotes in each (virtual) subpopulation.
- `homoNum_sp`: A dictionary of number of homozygotes in each (virtual) subpopulation.

genoFreq: This parameter accept a list of loci (by indexes or names) at which number and frequency of all genotypes are outputted as a dictionary (indexed by loci indexes) of default dictionaries (indexed by tuples of possible indexes). This statistic is available for all population types with genotype defined as ordered alleles at a locus. The length of genotype equals the number of homologous copies of chromosomes (ploidy) of a population. Genotypes for males or females on sex chromosomes or in haplodiploid populations will have different length. Because genotypes are ordered, (1, 0) and (0, 1) (two possible genotypes in a diploid population) are considered as different genotypes. This statistic outputs the following variables:

- `genoFreq` (default): A dictionary (by loci indexes) of default dictionaries (by genotype) of genotype frequencies. For example, `genoFreq[1][(1, 0)]` is the frequency of genotype (1, 0) at locus 1.
- `genoNum` (default): A dictionary of default dictionaries of genotype counts of all or specified (virtual) subpopulations.
- `genoFreq_sp`: genotype frequency in each specified (virtual) subpopulation.
- `genoNum_sp`: genotype count in each specified (virtual) subpopulation.

haploFreq: This parameter accepts one or more lists of loci (by index) at which number and frequency of haplotypes are outputted as default dictionaries. `[(1, 2)]` can be abbreviated to `(1, 2)`. For example, using parameter `haploFreq=(1, 2, 4)`, all haplotypes at loci 1, 2 and 4 are counted. This statistic saves results to dictionary (with loci index as keys) of default dictionaries (with haplotypes as keys) such as `haploFreq[(1, 2, 4)][(1, 1, 0)]` (frequency of haplotype (1, 1, 0) at loci (1, 2, 3)). This statistic works for all population types. Number of haplotypes for each individual equals to his/her ploidy number. Haplodiploid populations are supported in the sense that the second homologous copy of the haplotype is not counted for male individuals. This statistic outputs the following variables:

- `haploFreq` (default): A dictionary (with tuples of loci indexes as keys) of default dictionaries of haplotype frequencies. For example, `haploFreq[(0, 1)][(1, 1)]` records the frequency of haplotype (1, 1) at loci (0, 1) in all or specified (virtual) subpopulations.
- `haploNum` (default): A dictionary of default dictionaries of haplotype counts in all or specified (virtual) subpopulations.
- `haploFreq_sp`: Haptype frequencies in each (virtual) subpopulation.
- `haploNum_sp`: Halptype count in each (virtual) subpopulation.

haploHeteroFreq and **haploHomoFreq**: These parameters accept a list of haplotypes (list of loci), at which the number and frequency of haplotype homozygotes and/or heterozygotes will be calculated. Note that these statistics are **observed** count of haplotype heterozygote. The following variables will be outputted:

- `haploHeteroFreq` (default for parameter *haploHeteroFreq*): A dictionary of proportion of haplotype heterozygotes in all or specified (virtual) subpopulations, with haplotype indexes as dictionary keys.
- `haploHomoFreq` (default for parameter *haploHomoFreq*): A dictionary of proportion of homozygotes in all or specified (virtual) subpopulations.
- `haploHeteroNum`: A dictionary of number of heterozygotes in all or specified (virtual) subpopulations.
- `haploHomoNum`: A dictionary of number of homozygotes in all or specified (virtual) subpopulations.
- `haploHeteroFreq_sp`: A dictionary of proportion of heterozygotes in each (virtual) subpopulation.
- `haploHomoFreq_sp`: A dictionary of proportion of homozygotes in each (virtual) subpopulation.
- `haploHeteroNum_sp`: A dictionary of number of heterozygotes in each (virtual) subpopulation.
- `haploHomoNum_sp`: A dictionary of number of homozygotes in each (virtual) subpopulation.

sumOfInfo, meanOfInfo, varOfInfo, maxOfInfo and minOfInfo: Each of these five parameters accepts a list of information fields. For each information field, the sum, mean, variance, maximum or minimal (depending on the specified parameter(s)) of this information field at individuals in all or specified (virtual) subpopulations will be calculated. The results will be put into the following population variables:

- `sumOfInfo` (default for *sumOfInfo*): A dictionary of the sum of specified information fields of individuals in all or specified (virtual) subpopulations. This dictionary is indexed by names of information fields.
- `meanOfInfo` (default for *meanOfInfo*): A dictionary of the mean of information fields of all individuals.
- `varOfInfo` (default for *varOfInfo*): A dictionary of the sample variance of information fields of all individuals.
- `maxOfInfo` (default for *maxOfInfo*): A dictionary of the maximum value of information fields of all individuals.
- `minOfInfo` (default for *minOfInfo*): A dictionary of the minimal value of information fields of all individuals.
- `sumOfInfo_sp`: A dictionary of the sum of information fields of individuals in each subpopulation.
- `meanOfInfo_sp`: A dictionary of the mean of information fields of individuals in each subpopulation.
- `varOfInfo_sp`: A dictionary of the sample variance of information fields of individuals in each subpopulation.
- `maxOfInfo_sp`: A dictionary of the maximum value of information fields of individuals in each subpopulation.
- `minOfInfo_sp`: A dictionary of the minimal value of information fields of individuals in each subpopulation.

LD: Parameter LD accepts one or a list of loci pairs (e.g. `LD=[[0, 1], [2, 3]]`) with optional primary alleles at both loci (e.g. `LD=[0, 1, 0, 0]`). For each pair of loci, this operator calculates linkage disequilibrium and optional association statistics between two loci. When primary alleles are specified, signed linkage disequilibrium values are calculated with non-primary alleles are combined. Otherwise, absolute

values of diallelic measures are combined to yield positive measure of LD. Association measures are calculated from a m by n contingency of haplotype counts ($m=n=2$ if primary alleles are specified). Please refer to the simuPOP user's guide for detailed information. This statistic sets the following variables:

- LD (default) Basic LD measure for haplotypes in all or specified (virtual) subpopulations. Signed if primary alleles are specified.
- LD_prime (default) Lewontin's D' measure for haplotypes in all or specified (virtual) subpopulations. Signed if primary alleles are specified.
- R2 (default) Correlation LD measure for haplotypes in all or specified (virtual) subpopulations.
- LD_ChiSq ChiSq statistics for a contingency table with frequencies of haplotypes in all or specified (virtual) subpopulations.
- LD_ChiSq_p Single side p -value for the ChiSq statistic. Degrees of freedom is determined by number of alleles at both loci and the specification of primary alleles.
- CramerV Normalized ChiSq statistics.
- LD_sp Basic LD measure for haplotypes in each (virtual) subpopulation.
- LD_prime_sp Lewontin's D' measure for haplotypes in each (virtual) subpopulation.
- R2_sp R2 measure for haplotypes in each (virtual) subpopulation.
- LD_ChiSq_sp ChiSq statistics for each (virtual) subpopulation.
- LD_ChiSq_p_sp p value for the ChiSq statistics for each (virtual) subpopulation.
- CramerV_sp Cramer V statistics for each (virtual) subpopulation.

association: Parameter `association` accepts a list of loci, which can be a list of indexes, names, or ALL_AVAIL. At each locus, one or more statistical tests will be performed to test association between this locus and individual affection status. Currently, simuPOP provides the following tests:

- An allele-based Chi-square test using alleles counts. This test can be applied to loci with more than two alleles, and to haploid populations.
- A genotype-based Chi-square test using genotype counts. This test can be applied to loci with more than two alleles (more than 3 genotypes) in diploid populations. aA and Aa are considered to be the same genotype.
- A genotype-based Cochran-Armitage trend test. This test can only be applied to diallelic loci in diploid populations. A codominant model is assumed.

This statistic sets the following variables:

- Allele_ChiSq A dictionary of allele-based Chi-Square statistics for each locus, using cases and controls in all or specified (virtual) subpopulations.
- Allele_ChiSq_p (default) A dictionary of p -values of the corresponding Chi-square statistics.
- Geno_ChiSq A dictionary of genotype-based Chi-Square statistics for each locus, using cases and controls in all or specified (virtual) subpopulations.
- Geno_ChiSq_p A dictionary of p -values of the corresponding genotype-based Chi-square test.
- Armitage_p A dictionary of p -values of the Cochran- Armitage tests, using cases and controls in all or specified (virtual) subpopulations.
- Allele_ChiSq_sp A dictionary of allele-based Chi-Square statistics for each locus, using cases and controls from each subpopulation.
- Allele_ChiSq_p_sp A dictionary of p -values of allele-based Chi-square tests, using cases and controls from each (virtual) subpopulation.

- `Geno_ChiSq_sp` A dictionary of genotype-based Chi-Square tests for each locus, using cases and controls from each subpopulation.
- `Geno_ChiSq_p_sp` A dictionary of p-values of genotype-based Chi-Square tests, using cases and controls from each subpopulation.
- `Armitage_p_sp` A dictionary of *p-values* of the Cochran- Armitage tests, using cases and controls from each subpopulation.

neutrality: This parameter performs neutrality tests (detection of natural selection) on specified loci, which can be a list of loci indexes, names or `ALL_AVAIL`. It currently only outputs P_i , which is the average number of pairwise difference between loci. This statistic outputs the following variables:

- P_i Mean pairwise difference between all sequences from all or specified (virtual) subpopulations.
- P_{i_sp} Mean pairwise difference between all sequences in each (virtual) subpopulation.

structure: Parameter `structure` accepts a list of loci at which statistics that measure population structure are calculated. `structure` accepts a list of loci indexes, names or `ALL_AVAIL`. This parameter currently supports the following statistics:

- Weir and Cockerham's F_{st} (1984). This is the most widely used estimator of Wright's fixation index and can be used to measure Population differentiation. However, this method is designed to estimate F_{st} from samples of larger populations and might not be appropriate for the calculation of F_{st} of large populations.
- Nei's G_{st} (1973). The G_{st} estimator is another estimator for Wright's fixation index but it is extended for multi-allele (more than two alleles) and multi-loci cases. This statistics should be used if you would like to obtain a *true* F_{st} value of a large Population. Nei's G_{st} uses only allele frequency information so it is available for all population type (haploid, diploid etc). Weir and Cockerham's F_{st} uses heterozygosity frequency so it is best for autosome of diploid populations. For non-diploid population, sex, and mitochondrial DNAs, simuPOP uses expected heterozygosity ($1 - \sum p_i^2$) when heterozygosity is needed. These statistics output the following variables:
- F_{st} (default) The WC84 F_{st} statistic estimated for all * specified loci.
- F_{is} The WC84 F_{is} statistic estimated for all specified loci.
- F_{it} The WC84 F_{it} statistic estimated for all specified loci.
- f_{st} A dictionary of locus level WC84 F_{st} values.
- f_{is} A dictionary of locus level WC84 F_{is} values.
- f_{it} A dictionary of locus level WC84 F_{it} values.
- G_{st} Nei's G_{st} statistic estimated for all specified loci.
- g_{st} A dictionary of Nei's G_{st} statistic estimated for each locus.

HWE: Parameter `HWE` accepts a list of loci at which exact two-side tests for Hardy-Weinberg equilibrium will be performed. This statistic is only available for diallelic loci in diploid populations. `HWE` can be a list of loci indexes, names or `ALL_AVAIL`. This statistic outputs the following variables:

- `HWE` (default) A dictionary of p-values of HWE tests using genotypes in all or specified (virtual) subpopulations.
- `HWE_sp` A dictionary of p-values of HWS tests using genotypes in each (virtual) subpopulation.

inbreeding: Inbreeding measured by Identical by Decent (and by State). This statistics go through all loci of individuals in a diploid population and calculate the number and proportions of alleles that are identical by decent and by state. Because ancestral information is only available in lineage module, variables `IBD_freq` are always set to zero in other modules. Loci on sex and mitochondrial chromosomes, and non-diploid populations are currently not supported. This statistic outputs the following variables:

- `IBD_freq` (default) The frequency of IBD pairs among all allele pairs. To use this statistic, the population must be initialized by operator `InitLineage()` to assign each ancestral allele an unique identify.
- `IBS_freq` (default) The proportion of IBS pairs among all allele pairs.
- `IBD_freq_sp` frequency of IBD in each (virtual) subpopulations.
- `IBS_freq_sp` frequency of IBS in each (virtual) subpopulations.

effectiveSize: Parameter `effectiveSize` accepts a list of loci at which the effective population size for the whole or specified (virtual) subpopulations is calculated. *effectiveSize* can be a list of loci indexes, names or `ALL_AVAIL`. Parameter *subPops* is usually used to define samples from which effective sizes are estimated. This statistic allows the calculation of true effective size based on number of gametes each parents transmit to the offspring population (per- locus before and after mating), and estimated effective size based on sample genotypes. Due to the temporal natural of some methods, more than one Stat operators might be needed to calculate effective size. The *vars* parameter specified which method to use and which variable to set. Acceptable values include:

- `Ne_demo_base` When this variable is set before mating, it stores IDs of breeding parents and, more importantly, assign an unique lineage value to alleles at specified loci of each individual. **This feature is only available for lineage modules and will change lineage values at specified loci of all individuals.**
- `Ne_demo_base_sp` Pre-mating information for each (virtual) subpopulation, used by variable `Ne_demo_sp`.
- `Ne_demo` A dictionary of locus-specific demographic effective population size, calculated using number of gametes each parent transmits to the offspring population. The method is based on Crow & Denniston 1988 ($Ne = KN - 1/k - 1 + V_k/k$) and need variable `Ne_demo_base` set before mating. **Effective size estimated from this formula is model dependent and might not be applicable to your mating schemes.**
- `Ne_demo_sp` Calculate subpopulation-specific effective size.
- `Ne_temporal_base` When this variable is set in parameter *vars*, the Stat operator saves baseline allele frequencies and other information in this variable, which are used by temporary methods to estimate effective population size according to changes in allele frequency between the baseline and present generations. This variable could be set repeatedly to change baselines.
- `Ne_temporal_base_sp` Set baseline information for each (virtual) subpopulation specified.
- `Ne_tempoFS_P1` Effective population size, 2.5% and 97.5% confidence interval for sampling plan 1 as a list of size 3, estimated using a temporal method as described in Jorde & Ryman (2007), and as implemented by software `tempoFS` (<http://www.zoologi.su.se/~ryman/>). This variable is set to census population size if no baseline has been set, and to the temporal effective size between the present and the baseline generation otherwise. This method uses population size or sum of subpopulation sizes of specified (virtual) subpopulations as census population size for the calculation based on plan 1.
- `Ne_tempoFS_P2` Effective population size, 2.5% and 97.5% confidence interval for sampling plan 2 as a list of size 6, estimated using a temporal method as described in Jorde & Ryman (2007). This variable is set to census population size no baseline has been set, and to the temporal effective size between the present and the baseline generation otherwise. This method assumes that the sample is drawn from an infinitely-sized population.
- `Ne_tempoFS` deprecated, use `Ne_tempoFS_P2` instead.
- `Ne_tempoFS_P1_sp` Estimate effective size of each (virtual) subpopulation using method Jorde & Ryman 2007, assuming sampling plan 1. The census population sizes for sampling plan 1 are the sizes for each subpopulation that contain the specified (virtual) subpopulations.

- `Ne_tempoFS_P2_sp` Estimate effective size of each (virtual) subpopulation using method Jorde & Ryman 2007, assuming sampling plan 2.
- `Ne_tempoFS_sp` deprecated, use `Ne_tempoFS_P2_sp` instead.
- `Ne_waples89_P1` Effective population size, 2.5% and 97.5% confidence interval for sampling plan 1 as a list of size 6, estimated using a temporal method as described in Waples 1989, Genetics. Because this is a temporal method, `Ne_waples89` estimates effective size between the present and the baseline generation set by variable `Ne_temporal_base`. Census population size will be returned if no baseline has been set. This method uses population size or sum of subpopulation sizes of specified (virtual) subpopulations as census population size for the calculation based on plan 1.
- `Ne_waples89_P2` Effective population size, 2.5% and 97.5% confidence interval for sampling plan 2 as a list of size 6, estimated using a temporal method as described in Waples 1989, Genetics. Because this is a temporal method, `Ne_waples89` estimates effective size between the present and the baseline generation set by variable `Ne_temporal_base`. Census population size will be returned if no baseline has been set.
- `Ne_waples89_P1_sp` Estimate effective size for each (virtual) subpopulation using method Waples 89, assuming sampling plan 1. The census population sizes are the sizes for each subpopulation that contain the specified (virtual) subpopulation.
- `Ne_waples89_P2_sp` Estimate effective size for each (virtual) subpopulation using method Waples 89, assuming sampling plan 2.
- `Ne_waples89_sp` deprecated, use `Ne_waples89_P2_sp` instead.
- `Ne_LD` Lists of length three for effective population size, 2.5% and 97.% confidence interval for cutoff allele frequency 0., 0.01, 0.02 and 0.05 (as dictionary keys), using a parametric method, estimated from linkage disequilibrium information of one sample, using LD method developed by Waples & Do 2006 (LDNe). This method assumes unlinked loci and uses LD measured from genotypes at loci. Because this is a sample based method, it should better be applied to a random sample of the population. 95% CI is calculated using a Jackknife estimated effective number of independent alleles. Please refer to relevant papers and the LDNe user's guide for details.
- `Ne_LD_sp` Estimate LD-based effective population size for each specified (virtual) subpopulation.
- `Ne_LD_mono` A version of `Ne_LD` that assumes monogamy (see Waples 2006 for details).
- `Ne_LD_mono_sp` `Ne_LD_mono` calculated for each (virtual) subpopulation.

11.12 Conditional operators

11.12.1 class `IfElse`

class `IfElse`

This operator uses a condition, which can be a fixed condition, an expression or a user-defined function, to determine which operators to be applied when this operator is applied. A list of if-operators will be applied when the condition is `True`. Otherwise a list of else-operators will be applied.

`IfElse` (*cond*, *ifOps*=[], *elseOps*=[], *output*=">", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=`ALL_AVAIL`, *subPops*=`ALL_AVAIL`, *infoFields*=[])

Create a conditional operator that will apply operators *ifOps* if condition *cond* is met and *elseOps* otherwise. If a Python expression (a string) is given to parameter *cond*, the expression will be evaluated in each population's local namespace when this operator is applied. When a Python function is specified, it accepts parameter *pop* when it is applied to a population, and one or more parameters *pop*, *off*, *dad* or *mom* when it is applied during mating. The return value of this function should be `True` or `False`. Otherwise, parameter *cond* will be treated as a fixed condition (converted to `True` or `False`) upon which

one set of operators is always applied. The applicability of *ifOps* and *elseOps* are controlled by parameters *begin*, *end*, *step*, *at* and *rep* of both the *IfElse* operator and individual operators but *ifOps* and *elseOps* operators does not support negative indexes for replicate and generation numbers.

11.12.2 class TerminateIf

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. The generation at which the population is terminated will be counted in the *evolved generations* (return value from `Simulator::evolve`) if termination happens after mating.

TerminateIf (*condition=""*, *stopAll=False*, *message=""*, *output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create a terminator with an expression *condition*, which will be evaluated 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* (default to `""`), the generation number, proceeded with an optional *message*.

11.12.3 class DiscardIf

class DiscardIf

This operator discards individuals according to either an expression that evaluates according to individual information field, or a Python function that accepts individual and its information fields.

DiscardIf (*cond*, *exposeInd=""*, *output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an operator that discard individuals according to an expression or the return value of a Python function (parameter *cond*). This operator can be applied to a population before or after mating, or to offspring during mating. If an expression is passed to *cond*, it will be evaluated with each individual's information fields (see operator *InfoEval* for details). If *exposeInd* is non-empty, individuals will be available for evaluation in the expression as an variable with name spaced by *exposeInd*. If the expression is evaluated to be `True`, individuals (if applied before or after mating) or offspring (if applied during mating) will be removed or discard. Otherwise the return value should be either `False` (not discard), or a float number between 0 and 1 as the probability that the individual is removed. If a function is passed to *cond*, it should accept parameters *ind* and *pop* or names of information fields when it is applied to a population (pre or post mating), or parameters *off*, *dad*, *mom*, *pop* (parental population), or names of information fields if the operator is applied during mating. Individuals will be discarded if this function returns `True` or at a probability if a float number between 0 and 1 is returned. A constant expression (e.g. `True`, `False`, `0.4`) is also acceptable, with the last example (*cond=0.1*) that removes 10% of individuals at randomly. This operator supports parameter *subPops* and will remove only individuals belonging to specified (virtual) subpopulations.

11.13 The Python operator

11.13.1 class PyOperator

class PyOperator

An operator that calls a user-defined function when it is applied to a population (pre- or post-mating) or offspring (during- mating). The function can have parameters *pop* when the operator is applied pre- or

post-mating, `pop`, `off`, `dad`, `mom` when the operator is applied during-mating. An optional parameter can be passed if parameter `param` is given. In the during-mating case, parameters `pop`, `dad` and `mom` can be ignored if `offspringOnly` is set to `True`.

PyOperator (*func*, *param=None*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create a pure-Python operator that calls a user-defined function when it is applied. If this operator is applied before or after mating, your function should have form `func(pop)` or `func(pop, param)` where `pop` is the population to which the operator is applied, `param` is the value specified in parameter `param`. `param` will be ignored if your function only accepts one parameter. Alternatively, the function should have form `func(ind)` with optional parameters `pop` and `param`. In this case, the function will be called for all individuals, or individuals in subpopulations `subPops`. Individuals for which the function returns `False` will be removed from the population. This operator can therefore perform similar functions as operator `DiscardIf`.

If this operator is applied during mating, your function should accept parameters `pop`, `off` (or `ind`), `dad`, `mom` and `param` where `pop` is the parental population, and `off` or `ind`, `dad`, and `mom` are offspring and their parents for each mating event, and `param` is an optional parameter. If `subPops` are provided, only offspring in specified (virtual) subpopulations are acceptable.

This operator does not support parameters `output`, and `infoFields`. If certain output is needed, it should be handled in the user defined function `func`. Because the status of files used by other operators through parameter `output` is undetermined during evolution, they should not be open or closed in this Python operator.

11.14 Miscellaneous operators

11.14.1 class NoneOp

class NoneOp

This operator does nothing when it is applied to a population. It is usually used as a placeholder when an operator is needed syntactically.

NoneOp (*output=">"*, *begin=0*, *end=0*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)
Create a `NoneOp`.

11.14.2 class Dumper

class Dumper

This operator dumps the content of a population in a human readable format. Because this output format is not structured and can not be imported back to simuPOP, this operator is usually used to dump a small population to a terminal for demonstration and debugging purposes.

Dumper (*genotype=True*, *structure=True*, *ancGens=UNSPECIFIED*, *width=1*, *max=100*, *loci=[]*, *output=">"*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=ALL_AVAIL*)

Create a operator that dumps the genotype structure (if `structure` is `True`) and genotype (if `genotype` is `True`) to an `output` (default to standard terminal output). Because a population can be large, this operator will only output the first 100 (parameter `max`) individuals of the present generation (parameter `ancGens`). All loci will be outputted unless parameter `loci` are used to specify a subset of loci. This operator by default output values of all information fields unless parameter `infoFields` is used to specify a subset of info fields to display. If a list of (virtual) subpopulations are specified, this operator will only output individuals in

these outputs. Please refer to class *BaseOperator* for a detailed explanation for common parameters such as *output* and *stage*.

11.14.3 class SavePopulation

class SavePopulation

An operator that save populations to specified files.

SavePopulation (*output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an operator that saves a population to *output* when it is applied to the population. This operator supports all output specifications ('', 'filename', 'filename' prefixed by one or more '>' characters, and '!expr') but output from different operators will always replace existing files (effectively ignore '>' specification). Parameter *subPops* is ignored. Please refer to class *BaseOperator* for a detailed description about common operator parameters such as *stage* and *begin*.

11.14.4 class Pause

class Pause

This operator pauses the evolution of a simulator at given generations or at a key stroke. When a simulator is stopped, you can go to a Python shell to examine the status of an evolutionary process, resume or stop the evolution.

Pause (*stopOnKeyStroke=False*, *prompt=True*, *output=">"*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=ALL_AVAIL*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an operator that pause the evolution of a population when it is applied to this population. If *stopOnKeyStroke* is *False* (default), it will always pause a population when it is applied, if this parameter is set to *True*, the operator will pause a population if *any* key has been pressed. If a specific character is set, the operator will stop when this key has been pressed. This allows, for example, the use of several pause operators to pause different populations.

After a population has been paused, a message will be displayed (unless *prompt* is set to *False*) and tells you how to proceed. You can press 's' to stop the evolution of this population, 'S' to stop the evolution of all populations, or 'p' to enter a Python shell. The current population will be available in this Python shell as "pop_X_Y" when X is generation number and Y is replicate number. The evolution will continue after you exit this interactive Python shell.

Note: Ctrl-C will be intercepted even if a specific character is specified in parameter *stopOnKeyStroke*.

11.14.5 class TicToc

class TicToc

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. As an advanced feature that mainly used for performance testing, this operator accepts a parameter *stopAfter* (seconds), and will stop the evolution of a population if the overall time exceeds *stopAfter*. Note that elapsed time is only checked when this operator is applied to a population so it might not be able to stop the evolution process right after *stopAfter* seconds. This operator can also be applied during mating. Note that to avoid excessive time checking, this operator does not always check system time accurately.

TicToc (*output=">", stopAfter=0, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, sub-*
Pops=ALL_AVAIL, infoFields=[])

Create a *TicToc* operator that outputs the elapsed since the last time it was applied, and the overall time since the first time this operator is applied.

11.15 Function form of operators

11.15.1 Function acgtMutate

acgtMutate (*pop, *args, **kwargs*)

Function form of operator *AcgtMutator*

11.15.2 Function contextMutate

contextMutate (*pop, *args, **kwargs*)

Function form of operator *ContextMutator*

11.15.3 Function discardIf

discardIf (*pop, *args, **kwargs*)

Apply operator *DiscardIf* to population *pop* to remove individuals according to an expression or a Python function.

11.15.4 Function dump

dump (*pop, *args, **kwargs*)

Apply operator *Dumper* to population *pop*.

11.15.5 Function infoEval

infoEval (*pop, *args, **kwargs*)

Evaluate *expr* for each individual, using information fields as variables. Please refer to operator *InfoEval* for details.

11.15.6 Function infoExec

infoExec (*pop, *args, **kwargs*)

Execute *stmts* for each individual, using information fields as variables. Please refer to operator *InfoExec* for details.

11.15.7 Function initGenotype

initGenotype (*pop, *args, **kwargs*)

Apply operator *InitGenotype* to population *pop*.

11.15.8 Function `initInfo`

initInfo (*pop*, *args, **kwargs)

Apply operator *InitInfo* to population *pop*.

11.15.9 Function `initSex`

initSex (*pop*, *args, **kwargs)

Apply operator *InitSex* to population *pop*.

11.15.10 Function `kAlleleMutate`

kAlleleMutate (*pop*, *args, **kwargs)

Function form of operator *KAlleleMutator*

11.15.11 Function `maPenetrance`

maPenetrance (*pop*, *loci*, *penetrance*, *wildtype=0*, *ancGens=True*, *args, **kwargs)

Apply operator *MaPenetrance* to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

11.15.12 Function `mapPenetrance`

mapPenetrance (*pop*, *loci*, *penetrance*, *ancGens=True*, *args, **kwargs)

Apply operator *MapPenetrance* to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

11.15.13 Function `matrixMutate`

matrixMutate (*pop*, *args, **kwargs)

Function form of operator *MatrixMutator*

11.15.14 Function `mergeSubPops`

mergeSubPops (*pop*, *args, **kwargs)

Merge subpopulations *subPops* of population *pop* into a single subpopulation. Please refer to the operator form of this function (*MergeSubPops*) for details

11.15.15 Function `migrate`

migrate (*pop*, *args, **kwargs)

Function form of operator *Migrator*.

11.15.16 Function `backwardMigrate`

backwardMigrate (*pop*, *args, **kwargs)

Function form of operator *BackwardMigrator*.

11.15.17 Function mixedMutate

mixedMutate (*pop*, *args, **kwargs)
Function form of operator *MixedMutator*

11.15.18 Function mlPenetrance

mlPenetrance (*pop*, *ops*, *mode*, *ancGens=True*, *args, **kwargs)
Apply operator *MapPenetrance* to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

11.15.19 Function pointMutate

pointMutate (*pop*, *args, **kwargs)
Function form of operator *PointMutator*

11.15.20 Function pyEval

pyEval (*pop*, *args, **kwargs)
Evaluate statements *stmts* (optional) and expression *expr* in population *pop*'s local namespace and return the result of *expr*. If *exposePop* is given, population *pop* will be exposed in its local namespace as a variable with a name specified by *exposePop*. Unlike its operator counterpart, this function returns the result of *expr* rather than writing it to an output.

11.15.21 Function pyExec

pyExec (*pop*, *args, **kwargs)
Execute *stmts* in population *pop*'s local namespace.

11.15.22 Function pyMutate

pyMutate (*pop*, *args, **kwargs)
Function form of operator *PyMutator*

11.15.23 Function pyPenetrance

pyPenetrance (*pop*, *func*, *loci=[]*, *ancGens=True*, *args, **kwargs)
Apply operator *PyPenetrance* to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

11.15.24 Function pyMlPenetrance

pyMlPenetrance (*pop*, *func*, *mode*, *loci=[]*, *ancGens=True*, *args, **kwargs)
Apply operator *PyMlPenetrance* to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

11.15.25 Function `pyQuanTrait`

pyQuanTrait (*pop*, *func*, *loci*=[], *ancGens*=True, **args*, ***kwargs*)

Apply operator `PyQuanTrait` to population *pop*. Unlike the operator form of this operator that only handles the current generation, this function by default assign affection status to all generations.

11.15.26 Function `resizeSubPops`

resizeSubPops (*pop*, **args*, ***kwargs*)

Resize subpopulations *subPops* of population *pop* into new sizes *size*. Individuals will be added or removed accordingly. Please refer to the operator form of this function (`ResizeSubPops`) for details

11.15.27 Function `snpMutate`

snpMutate (*pop*, **args*, ***kwargs*)

Function form of operator `SNPMutator`

11.15.28 Function `splitSubPops`

splitSubPops (*pop*, **args*, ***kwargs*)

Split subpopulations (*subPops*) of population *pop* according to either *sizes* or *proportions* of the resulting subpopulations, or an information field. Please refer to the operator form of this function (`splitSubPop`) for details.

11.15.29 Function `stat`

stat (*pop*, **args*, ***kwargs*)

Apply operator `Stat` with specified parameters to population *pop*. Resulting statistics could be accessed from the local namespace of *pop* using functions `pop.vars()` or `pop.dvars()`

11.15.30 Function `stepwiseMutate`

stepwiseMutate (*pop*, **args*, ***kwargs*)

Function form of operator `StepwiseMutator`

11.15.31 Function `tagID`

tagID (*pop*, *reset*=False, **args*, ***kwargs*)

Apply operator `IdTagger` to population *pop* to assign a unique ID to all individuals in the population. Individuals ID will starts from a system wide index. You can reset this start ID using parameter *reset* which can be `True` (reset to 1) or a non-negative number (start from this number).

12.1 Module `simuOpt`

Module `simuOpt` provides a function `simuOpt.setOptions` to control which `simuPOP` module to load, and how it is loaded, and a class `simuOpt.Params` that helps users manage simulation parameters.

When `simuPOP` is loaded, it checks a few environmental variables (`SIMUOPTIMIZED`, `SIMUALLELETYPE`, and `SIMUDEBUG`) to determine which `simuPOP` module to load, and how to load it. More options can be set using the `simuOpt.setOptions` function. For example, you can suppress the banner message when `simuPOP` is loaded and require a minimal version of `simuPOP` for your script. `simuPOP` recognize the following commandline arguments

--optimized Load the optimized version of a `simuPOP` module.

--gui=None|batch|interactive|True|wxPython|Tkinter Whether or not use a graphical toolkit and which one to use. `--gui=batch` is usually used to run a script in batch mode (do not start a parameter input dialog and use all default values unless a parameter is specified from command line or a configuration file. If `--gui=interactive`, an interactive shell will be used to solicit input from users. Otherwise, `simuPOP` will try to use a graphical parameter input dialog, and falls to an interactive mode when no graphical Toolkit is available. Please refer to parameter `gui` for `simuOpt.setOptions` for details.

class `params.Params` provides a powerful way to handle commandline arguments. Briefly speaking, a `Params` object can be created from a list of parameter specification dictionaries. The parameters are then become attributes of this object. A number of functions are provided to determine values of these parameters using commandline arguments, a configuration file, or a parameter input dialog (using `Tkinter` or `wxPython`). Values of these parameters can be accessed as attributes, or extracted as a list or a dictionary. Note that the `Params.getParam` function automatically handles the following commandline arguments.

-h or --help Print usage message.

--config=configFile Read parameters from a configuration file *configFile*.

12.1.1 Function `setOptions`

`simuOpt.setOptions` (*alleleType=None, optimized=None, gui=None, quiet=None, debug=None, version=None, revision=None, numThreads=None, plotter=None*)

Set options before `simuPOP` is loaded to control which `simuPOP` module to load, and how the module should be loaded.

alleleType Use the standard, binary, long or mutant allele version of the `simuPOP` module if `alleleType` is set to 'short', 'binary', 'long', 'mutant', or 'lineage' respectively. If this parameter is not set, this function will try to get its value from environmental variable `SIMUALLELETYPE`. The standard (short) module will be used if the environmental variable is not defined.

optimized Load the optimized version of a module if this parameter is set to `True` and the standard version if it is set to `False`. If this parameter is not set (`None`), the optimized version will be used if environmental variable `SIMUOPTIMIZED` is defined. The standard version will be used otherwise.

gui Whether or not use graphical user interfaces, which graphical toolkit to use and how to process parameters in non-GUI mode. If this parameter is `None` (default), this function will check environmental variable `SIMUGUI` or commandline option `--gui` for a value, and assume `True` if such an option is unavailable. If `gui=True`, `simuPOP` will use `wxPython`-based dialogs if `wxPython` is available, and use `Tkinter`-based dialogs if `Tkinter` is available and use an interactive shell if no graphical toolkit is available. `gui='Tkinter'` or `'wxPython'` can be used to specify the graphical toolkit to use. If `gui='interactive'`, a `simuPOP` script prompt users to input values of parameters. If `gui='batch'`, default values of unspecified parameters will be used. In any case, commandline arguments and a configuration file specified by parameter `-config` will be processed. This option is usually left to `None` so that the same script can be run in both GUI and batch mode using commandline option `--gui`.

plotter (Deprecated)

quiet If set to `True`, suppress the banner message when a `simuPOP` module is loaded.

debug A list of debug code (as string) that will be turned on when `simuPOP` is loaded. If this parameter is not set, a list of comma separated debug code specified in environmental variable `SIMUDEBUG`, if available, will be used. Note that setting `debug=[]` will remove any debug code that might have been by variable `SIMUDEBUG`.

version A version string (e.g. 1.0.0) indicating the required version number for the `simuPOP` module to be loaded. `simuPOP` will fail to load if the installed version is older than the required version.

revision Obsolete with the introduction of parameter `version`.

numThreads Number of Threads that will be used to execute a `simuPOP` script. The values can be a positive number (number of threads) or 0 (all available cores of the computer, or whatever number set by environmental variable `OMP_NUM_THREADS`). If this parameter is not set, the number of threads will be set to 1, or a value set by environmental variable `OMP_NUM_THREADS`.

12.2 Module `simuPOP.utils`

This module provides some commonly used operators and format conversion utilities.

12.2.1 class `Trajectory`

class `simuPOP.utils.Trajectory`

A `Trajectory` object contains frequencies of one or more loci in one or more subpopulations over several

generations. It is usually returned by member functions of class `TrajectorySimulator` or equivalent global functions `simulateForwardTrajectory` and `simulateBackwardTrajectory`.

The `Trajectory` object provides several member functions to facilitate the use of Trajectory-simulation techniques. For example, `Trajectory.func()` returns a trajectory function that can be provided directly to a `ControlledOffspringGenerator`; `Trajectory.mutators()` provides a list of `PointMutator` that insert mutants at the right generations to initialize a trajectory.

For more information about Trajectory simulation techniques and related controlled random mating scheme, please refer to the simuPOP user's guide, and Peng et al (PLoS Genetics 3(3), 2007).

Trajectory (*endGen*, *nLoci*)

Create a `Trajectory` object of alleles at *nLoci* loci with ending generation *endGen*. *endGen* is the generation when expected allele frequencies are reached after mating. Therefore, a trajectory for 1000 generations should have `endGen=999`.

freq (*gen*, *subPop*)

Return frequencies of all loci in subpopulation *subPop* at generation *gen* of the simulated Trajectory. Allele frequencies are assumed to be zero if *gen* is out of range of the simulated Trajectory.

func ()

Return a Python function that returns allele frequencies for each locus at specified loci. If there are multiple subpopulations, allele frequencies are arranged in the order of `loc0_sp0`, `loc1_sp0`, ..., `loc0_sp1`, `loc1_sp1`, ... and so on. The returned function can be supplied directly to the `freqFunc` parameter of a controlled random mating scheme (`ControlledRandomMating`) or a homogeneous mating scheme that uses a controlled offspring generator (`ControlledOffspringGenerator`).

mutants ()

Return a list of mutants in the form of (`loc`, `gen`, `subPop`)

mutators (*loci*, *inds=0*, *allele=1*, **args*, ***kwargs*)

Return a list of `PointMutator` operators that introduce mutants at the beginning of simulated trajectories. These mutators should be added to the `preOps` parameter of `Simulator.evolve` function to introduce a mutant at the beginning of a generation with zero allele frequency before mating, and a positive allele frequency after mating. A parameter `loci` is needed to specify actual loci indexes in the real forward simulation. Other than default parameters `inds=0` and `allele=1`, additional parameters could be passed to point mutator as keyword parameters.

12.2.2 class TrajectorySimulator

class `simuPOP.utils.TrajectorySimulator`

A Trajectory Simulator takes basic demographic and genetic (natural selection) information of an evolutionary process of a diploid population and allow the simulation of Trajectory of allele frequencies of one or more loci. Trajectories could be simulated in two ways: forward-time and backward-time. In a forward-time simulation, the simulation starts from certain allele frequency and simulate the frequency at the next generation using given demographic and genetic information. The simulation continues until an ending generation is reached. A Trajectory is successfully simulated if the allele frequency at the ending generation falls into a specified range. In a backward-time simulation, the simulation starts from the ending generation with a desired allele frequency and simulate the allele frequency at previous generations one by one until the allele gets lost (allele frequency equals zero).

The result of a trajectory simulation is a trajectory object which can be used to direct the simulation of a special random mating process that controls the evolution of one or more disease alleles so that allele frequencies are consistent across replicate simulations. For more information about Trajectory simulation techniques and related controlled random mating scheme, please refer to the simuPOP user's guide, and Peng et al (PLoS Genetics 3(3), 2007).

TrajectorySimulator (*N*, *nLoci*=1, *fitness*=None, *logger*=None)

Create a trajectory Simulator using provided demographic and genetic (natural selection) parameters. Member functions *simuForward* and *simuBackward* can then be used to simulate trajectories within certain range of generations. This class accepts the following parameters

N Parameter *N* accepts either a constant number for population size (e.g. *N*=1000), a list of subpopulation sizes (e.g. *N*=[1000, 2000]), or a demographic function that returns population or subpopulation sizes at each generation. During the evolution, multiple subpopulations can be merged into one, and one population can be split into several subpopulations. The number of subpopulation is determined by the return value of the demographic function. Note that *N* should be considered as the population size at the end of specified generation.

nLoci Number of unlinked loci for which trajectories of allele frequencies are simulated. We assume a diploid population with diallelic loci. The Trajectory represents frequencies of a

fitness Parameter *fitness* can be None (no selection), a list of fitness values for genotype with 0, 1, and 2 disease alleles (*AA*, *Aa*, and *aa*) at one or more loci; or a function that returns fitness values at each generation. When multiple loci are involved (*nLoci*), *fitness* can be a list of 3 (the same fitness values for all loci), a list of 3**nLoci* (different fitness values for each locus) or a list of 3***nLoci* (fitness value for each combination of genotype). The fitness function should accept generation number and a subpopulation index. The latter parameter allows, and is the only way to specify different fitness in each subpopulation.

logger A logging object (see Python module *logging*) that can be used to output intermediate results with debug information.

simuBackward (*endGen*, *endFreq*, *minMutAge*=None, *maxMutAge*=None, *maxAttempts*=1000)

Simulate trajectories of multiple disease susceptibility loci using a forward time approach. This function accepts allele frequencies of alleles of multiple unlinked loci (*endFreq*) at the end of generation *endGen*. Depending on the number of loci and subpopulations, parameter *beginFreq* can be a number (same frequency for all loci in all subpopulations), or a list of frequencies for each locus (same frequency in all subpopulations), or a list of frequencies for each locus in each subpopulation in the order of *loc0_sp0*, *loc1_sp0*, ..., *loc0_sp1*, *loc1_sp1*, ... and so on.

This simulator will simulate a trajectory generation by generation and restart if the disease allele got fixed (instead of lost), or if the length simulated Trajectory does not fall into *minMutAge* and *maxMutAge* (ignored if None is given). This simulator will return None if no valid Trajectory is found after *maxAttempts* attempts.

simuForward (*beginGen*, *endGen*, *beginFreq*, *endFreq*, *maxAttempts*=10000)

Simulate trajectories of multiple disease susceptibility loci using a forward time approach. This function accepts allele frequencies of alleles of multiple unlinked loci at the beginning generation (*freq*) at generation *beginGen*, and expected *range* of allele frequencies of these alleles (*endFreq*) at the end of generation *endGen*. Depending on the number of loci and subpopulations, these parameters accept the following inputs:

beginGen Starting generation. The initial frequencies are considered as frequencies at the *beginning* of this generation.

endGen Ending generation. The ending frequencies are considered as frequencies at the *end* of this generation.

beginFreq The initial allele frequency of involved loci in all subpopulations. It can be a number (same frequency for all loci in all subpopulations), or a list of frequencies for each locus (same frequency in all subpopulations), or a list of frequencies for each locus in each subpopulation in the order of *loc0_sp0*, *loc1_sp0*, ..., *loc0_sp1*, *loc1_sp1*, ... and so on.

endFreq The range of acceptable allele frequencies at the ending generation. The ranges can be specified for all loci in all subpopulations, for all loci (allele frequency in the whole population is considered),

or for all loci in all subpopulations, in the order of `loc0_sp0`, `loc1_sp0`, ..., `loc0_sp1`, ... and so on.

This simulator will simulate a trajectory generation by generation and restart if the resulting frequencies do not fall into specified range of frequencies. This simulator will return `None` if no valid `Trajectory` is found after `maxAttempts` attempts.

12.2.3 Function `simulateForwardTrajectory`

```
simuPOP.utils.simulateForwardTrajectory(N, beginGen, endGen, beginFreq, endFreq,
                                         nLoci=1, fitness=None, maxAttempts=10000,
                                         logger=None)
```

Given a demographic model (*N*) and the fitness of genotype at one or more loci (*fitness*), this function simulates a trajectory of one or more unlinked loci (*nLoci*) from allele frequency *freq* at generation *beginGen* forward in time, until it reaches generation *endGen*. A `Trajectory` object will be returned if the allele frequency falls into specified ranges (*endFreq*). `None` will be returned if no valid `Trajectory` is simulated after `maxAttempts` attempts. Please refer to class `Trajectory`, `TrajectorySimulator` and their member functions for more details about allowed input for these parameters. If a *logger* object is given, it will send detailed debug information at `DEBUG` level and ending allele frequencies at the `INFO` level. The latter can be used to adjust your fitness model and/or ending allele frequency if a trajectory is difficult to obtain because of parameter mismatch.

12.2.4 Function `simulateBackwardTrajectory`

```
simuPOP.utils.simulateBackwardTrajectory(N, endGen, endFreq, nLoci=1, fitness=None,
                                          minMutAge=None, maxMutAge=None, maxAttempts=1000, logger=None)
```

Given a demographic model (*N*) and the fitness of genotype at one or more loci (*fitness*), this function simulates a trajectory of one or more unlinked loci (*nLoci*) from allele frequency *freq* at generation *endGen* backward in time, until all alleles get lost. A `Trajectory` object will be returned if the length of simulated `Trajectory` with *minMutAge* and *maxMutAge* (if specified). `None` will be returned if no valid `Trajectory` is simulated after `maxAttempts` attempts. Please refer to class `Trajectory`, `TrajectorySimulator` and their member functions for more details about allowed input for these parameters. If a *logger* object is given, it will send detailed debug information at `DEBUG` level and ending generation and frequency at the `INFO` level. The latter can be used to adjust your fitness model and/or ending allele frequency if a trajectory is difficult to obtain because of parameter mismatch.

12.2.5 class `ProgressBar`

```
class simuPOP.utils.ProgressBar
```

The `ProgressBar` class defines a progress bar. This class will use a text-based progress bar that outputs progressing dots (.) with intermediate numbers (e.g. 5 for 50%) under a non-GUI mode (*gui*=`False`) or not displaying any progress bar if *gui*=`'batch'`. In the GUI mode, a Tkinter or wxPython progress dialog will be used (*gui*=`Tkinter` or *gui*=`wxPython`). The default mode is determined by the global *gui* mode of `simuPOP` (see also `simuOpt.setOptions`).

This class is usually used as follows:

```
progress = ProgressBar("Start simulation", 500)
for i in range(500):
    # i+1 can be ignored if the progress bar is updated by 1 step
    progress.update(i+1)
# if you would like to make sure the done message is displayed.
progress.done()
```

ProgressBar (*message*, *totalCount*, *progressChar*='.', *block*=2, *done*=' Done.n', *gui*=None)

Create a progress bar with *message*, which will be the title of a progress dialog or a message for textbased progress bar. Parameter *totalCount* specifies total expected steps. If a text-based progress bar is used, you could specified progress character and intervals at which progresses will be displayed using parameters *progressChar* and *block*. A ending message will also be displayed in text mode.

done ()

Finish progressbar, print 'done' message if in text-mode.

update (*count*=None)

Update the progreebar with *count* steps done. The dialog or textbar may not be updated if it is updated by full percent(s). If *count* is None, the progressbar increases by one step (not percent).

12.2.6 Function viewVars

`simuPOP.utils.viewVars` (*var*, *gui*=None)

list a variable in tree format, either in text format or in a wxPython window.

var A dictionary variable to be viewed. Dictionary wrapper objects returned by `Population.dvars()` and `Simulator.dvars()` are also acceptable.

gui If *gui* is False or 'Tkinter', a text presentation (use the pprint module) of the variable will be printed to the screen. If *gui* is 'wxPython' and wxPython is available, a wxPython windows will be used. The default mode is determined by the global gui mode (see also `simuOpt.setOptions`).

12.2.7 Function saveCSV

`simuPOP.utils.saveCSV` (*pop*, *filename*="", *infoFields*=[], *loci*=True, *header*=True, *subPops*=ALL_AVAIL, *genoFormatter*=None, *infoFormatter*=None, *sexFormatter*={1: 'M', 2: 'F'}, *affectionFormatter*={True: 'A', False: 'U'}, *sep*='.', **kwargs)

This function is deprecated. Please use `export(format='csv')` instead. Save a simuPOP population *pop* in csv format. Columns of this file is arranged in the order of information fields (*infoFields*), sex (if *sexFormatter* is not None), affection status (if *affectionFormatter* is not None), and genotype (if *genoFormatter* is not None). This function only output individuals in the present generation of population *pop*. This function accepts the following parameters:

pop A simuPOP population object.

filename Output filename. Leading '>' characters are ignored. However, if the first character of this filename is '!', the rest of the name will be evaluated in the population's local namespace. If *filename* is empty, the content will be written to the standard output.

infoFileds Information fields to be outputted. Default to none.

loci If a list of loci is given, only genotype at these loci will be written. Default to ALL_AVAIL, meaning all available loci. You can set this parameter to [] if you do not want to output any genotype.

header Whether or not a header should be written. These headers will include information fields, sex (if *sexFormatter* is not None), affection status (if *affectionFormatter* is not None) and loci names. If genotype at a locus needs more than one column, *_1*, *_2* etc will be appended to loci names. Alternatively, a complete header (a string) or a list of column names could be specified directly.

subPops A list of (virtual) subpopulations. If specified, only individuals from these subpopulations will be outputted.

infoFormatter A format string that is used to format all information fields. If unspecified, `str(value)` will be used for each information field.

genoFormatter How to output genotype at specified loci. Acceptable values include `None` (output allele names), a dictionary with genotype as keys, (e.g. `genoFormatter={(0,0):1, (0,1):2, (1,0):2, (1,1):3}`), or a function with genotype (as a tuple of integers) as inputs. The dictionary value or the return value of this function can be a single or a list of number or strings.

sexFormatter How to output individual sex. Acceptable values include `None` (no output) or a dictionary with keys `MALE` and `FEMALE`.

affectionFormatter How to output individual affection status. Acceptable values include `None` (no output) or a dictionary with keys `True` and `False`.

Parameters `genoCode`, `sexCode`, and `affectionCode` from version 1.0.0 have been renamed to `genoFormatter`, `sexFormatter` and `affectionFormatter` but can still be used.

12.2.8 class `Exporter`

class `simuPOP.utils.Exporter`

An operator to export the current population in specified format. Currently supported file formats include:

STRUCTURE (<http://pritch.bsd.uchicago.edu/structure.html>). This format accepts the following parameters:

markerNames If set to `True` (default), output names of loci that are specified by parameter `lociNames` of the `Population` class. No names will be outputted if loci are anonymous. A list of loci names are acceptable which will be outputted directly.

recessiveAlleles If specified, value of this parameter will be outputted after the marker names line.

interMarkerDistances If set to `True` (default), output distances between markers. The first marker of each chromosome has distance -1, as required by this format.

phaseInformation If specified, output the value (0 or 1) of this parameter after the inter marker distances line. Note that `simuPOP` populations always have phase information.

label Output 1-based indexes of individuals if this parameter is true (default)

popData Output 1-based index of subpopulation if this parameter is set to true (default).

popFlag Output value of this parameter (0 or 1) after `popData` if this parameter specified.

locData Name of an information field with location information of each individual. Default to `None` (no location data)

phenotype Name of an information field with phenotype information of each individual. Default to `None` (no phenotype)

Genotype information are always outputted. Alleles are coded the same way (0, 1, 2, etc) as they are stored in `simuPOP`.

GENEPOP (<http://genepop.curtin.edu.au/>). The `genepop` format accepts the following parameters:

title The title line. If unspecified, a line similar to ‘produced by `simuPOP` on XXX’ will be outputted.

adjust Adjust values of alleles by specified value (1 as default). This adjustment is necessary in many cases because `GENEPOP` treats allele 0 as missing values, and `simuPOP` treats allele 0 as a valid allele. Exporting alleles 0 and 1 as 1 and 2 will allow `GENEPOP` to analyze `simuPOP`-exported files correctly.

Because 0 is reserved as missing data in this format, allele A is outputted as A+adjust. `simuPOP` will use subpopulation names (if available) and 1-based individual index to output individual label (e.g. SubPop2-3). If

parameter `subPops` is used to output selected individuals, each subpop will be outputted as a separate subpopulation even if there are multiple virtual subpopulations from the same subpopulation. `simuPOP` currently only export diploid populations to this format.

FSTAT (<http://www2.unil.ch/popgen/softwares/fstat.htm>). The `fstat` format accepts the following parameters:

lociNames Names of loci that will be outputted. If unspecified, `simuPOP` will try to use names of loci that are specified by parameter `lociNames` of the `Population` class, or names in the form of `chrX-Y`.

adjust Adjust values of alleles by specified value (1 as default). This adjustment is necessary in many cases because FSTAT treats allele 0 as missing values, and `simuPOP` treats allele 0 as a valid allele. Exporting alleles 0 and 1 as 1 and 2 will allow FSTAT to analyze `simuPOP`-exported files correctly.

MAP (marker information format) output information about each loci. Each line of the map file describes a single marker and contains chromosome name, locus name, and position. Chromosome and loci names will be the names specified by parameters `chromNames` and `lociNames` of the `Population` object, and will be chromosome index + 1, and '.' if these parameters are not specified. This format output loci position to the third column. If the unit assumed in your population does not match the intended unit in the MAP file, (e.g. you would like to output position in basepair while the population uses Mbp), you can use parameter `posMultiplier` to adjust it. This format accepts the following parameters:

posMultiplier A number that will be multiplied to loci positions (default to 1). The result will be outputted in the third column of the output.

PED (Linkage Pedigree pre MAKEPED format), with columns of family, individual, father mother, gender, affection status and genotypes. The output should be acceptable by HaploView or plink, which provides more details of this format in their documentation. If a population does not have `ind_id`, `father_id` or `mother_id`, this format will output individuals in specified (virtual) subpopulations in the current generation (parental generations are ignored) as unrelated individuals with 0, 0 as parent IDs. An incremental family ID will be assigned for each individual. If a population have `ind_id`, `father_id` and `mother_id`, parents will be recursively traced to separate all individuals in a (multigenerational) population into families of related individuals. father and mother id will be set to zero if one of them does not exist. This format uses 1 for MALE, 2 for FEMALE. If `phenoField` is `None`, individual affection status will be outputted with 1 for Unaffected and 2 for affected. Otherwise, values of an information field will be outputted as phenotype. Because 0 value indicates missing value, values of alleles will be adjusted by 1 by default, which should be avoided if you are using non-zero alleles to model ACTG alleles in `simuPOP`. This format will ignore subpopulation structure because parents might belong to different subpopulations. This format accepts the following parameters:

idField A field for individual id, default to `ind_id`. Value at this field will be individual ID inside a pedigree.

fatherField A field for father id, default to `father_id`. Value at this field will be used to output father of an individual, if an individual with this ID exists in the population.

motherField A field for mother id, default to `mother_id`. Value at this field will be used to output mother of an individual, if an individual with this ID exists in the population.

phenoField A field for individual phenotype that will be outputted as the sixth column of the PED file. If `None` is specified (default), individual affection status will be outputted (1 for unaffected and 2 for affected).

adjust Adjust values of alleles by specified value (1 as default). This adjustment is necessary in many cases because LINKAGE/PED format treats allele 0 as missing values, and `simuPOP` treats allele 0 as a valid allele. You should set this parameter to zero if you have already used alleles 1, 2, 3, 4 to model A, C, T, and G alleles.

Phylip (Joseph Felsenstein's Phylip format). Phylip is generally used for nucleotide sequences and protein sequences. This makes this format suitable for simulations of haploid populations (`ploidy=1`) with nucleotide or protein sequences (number of alleles = 4 or 24 with `alleleNames` as nucleotide or amino acid names). If your population does satisfy these conditions, you can still export it, with homologous chromosomes in a diploid population as two sequences, and with specified allele names for allele 0, 1, 2, ... This function outputs sequence

name as SXXX where XXX is the 1-based index of individual and SXXX_Y (Y=1 or 2) for diploid individuals, unless names of sequences are provided by parameter `seqNames`. This format supports the following parameters:

alleleNames Names of alleles 0, 1, 2, ... as a single string (e.g. 'ACTG') or a list of single-character strings (e.g. ['A', 'C', 'T', 'G']). If this parameter is unspecified (default), this program will try to use names of alleles specified in `alleleNames` parameter of a Population, and raise an error if no name could be found.

seqNames Names of each sequence outputted, for each individual, or for each sequences for non-haploid population. If unspecified, default names such as SXXX or SXXX_Y will be used.

style Output style, can be 'sequential' (default) or 'interleaved'. For sequential output, each sequence consists of for the first line a name and 90 symbols starting from column 11, and subsequent lines of 100 symbols. The interleaved style have subsequent lines as separate blocks.

MS (output from Richard R. Hudson's MS or msHOT program). This format records genotypes of SNP markers at segregating site so all non-zero genotypes are recorded as 1. simuPOP by default outputs a single block of genotypes at all loci on the first chromosome, and for all individuals, unless parameter `splitBy` is specified to separate genotypes by chromosome or subpopulations.

splitBy: simuPOP by default output segregating sites at all loci on the first chromosome for all individuals. If `splitBy` is set to 'subPop', genotypes for individuals in all or specified (parameter `subPops`) subpopulations are outputted in separate blocks. The subpopulations should have the same number of individuals to produce blocks of the same number of sequences. Alternatively, `splitBy` can be set to `chrom`, for which genotypes on different chromosomes will be outputted separately.

CSV (comma separated values). This is a general format that output genotypes in comma (or tab etc) separated formats. The function form of this operator `export(format='csv')` is similar to the now-deprecated `saveCSV` function, but its interface has been adjusted to match other formats supported by this operator. This format outputs a header (optional), and one line for each individual with values of specified information fields, sex, affection status, and genotypes. All fields except for genotypes are optional. The output format is controlled by the following parameters:

infoFields Information fields to be outputted. Default to none.

header Whether or not a header should be written. These headers will include information fields, sex (if `sexFormatter` is not None), affection status (if `affectionFormatter` is not None) and loci names. If genotype at a locus needs more than one column, `_1`, `_2` etc will be appended to loci names. Alternatively, a complete header (a string) or a list of column names could be specified directly.

infoFormatter A format string that is used to format all information fields. If unspecified, `str(value)` will be used for each information field.

genoFormatter How to output genotype at specified loci. Acceptable values include None (output allele values), a dictionary with genotype as keys, (e.g. `genoFormatter={ (0,0):1, (0,1):2, (1,0):2, (1,1):3 }`), or a function with genotype (as a tuple of integers) as inputs. The dictionary value or the return value of this function can be a single or a list of number or strings.

sexFormatter How to output individual sex. Acceptable values include None (no output) or a dictionary with keys MALE and FEMALE.

affectionFormatter How to output individual affection status. Acceptable values include None (no output) or a dictionary with keys True and False.

delimiter Delimiter used to separate values, default to ','.

subPopFormatter How to output population membership. Acceptable values include None (no output), a string that will be used for the column name, or True which uses 'pop' as the column name. If present, the column is written with the string representation of the (virtual) subpopulation.

This operator supports the usual applicability parameters such as `begin`, `end`, `step`, `at`, `reps`, and `subPops`. If `subPops` are specified, only individuals from specified (virtual) `subPops` are exported. Similar to other operators,

parameter `output` can be an output specification string (`filename`, `>>filename`, `!expr`), `filehandle` (or any Python object with a `write` function), any python function. Unless explicitly stated for a particular format, this operator exports individuals from the current generation if there are multiple ancestral generations in the population.

The Exporter class will make use of a progress bar to show the progress. The interface of the progress bar is by default determined by the global GUI status but you can also set it to, for example, `gui=False` to forcefully use a text-based progress bar, or `gui='batch'` to suppress the progress bar.

Exporter (*format, output, begin=0, end=-1, step=1, at=[], reps=True, subPops=ALL_AVAIL, infoFields=[], gui=None, *args, **kwargs*)

Usage:

PyOperator(*func, param=None, begin=0, end=-1, step=1, at=[], reps=ALL_AVAIL, subPops=ALL_AVAIL, infoFields=[]*)

Details:

Create a pure-Python operator that calls a user-defined function when it is applied. If this operator is applied before or after mating, your function should have form `func(pop)` or `func(pop, param)` where `pop` is the population to which the operator is applied, `param` is the value specified in parameter `param`. `param` will be ignored if your function only accepts one parameter. Alternatively, the function should have form `func(ind)` with optional parameters `pop` and `param`. In this case, the function will be called for all individuals, or individuals in subpopulations `subPops`. Individuals for which the function returns `False` will be removed from the population. This operator can therefore perform similar functions as operator `DiscardIf`. If this operator is applied during mating, your function should accept parameters `pop`, `off` (or `ind`), `dad`, `mom` and `param` where `pop` is the parental population, and `off` or `ind`, `dad`, and `mom` are offspring and their parents for each mating event, and `param` is an optional parameter. If `subPops` are provided, only offspring in specified (virtual) subpopulations are acceptable. This operator does not support parameters `output`, and `infoFields`. If certain output is needed, it should be handled in the user defined function `func`. Because the status of files used by other operators through parameter `output` is undetermined during evolution, they should not be open or closed in this Python operator.

12.2.9 Function `importPopulation`

`simuPOP.utils.importPopulation` (*format, filename, *args, **kwargs*)

This function import and return a population from a file `filename` in specified `format`. Format-specific parameters can be used to define how the input should be interpreted and imported. This function supports the following file format.

GENEPOP (<http://genepop.curtin.edu.au/>). For input file of this format, this function ignores the first title line, load the second line as loci names, and import genotypes of different POP sections as different subpopulations. This format accepts the following parameters:

adjust Adjust alleles by specified value (default to 0 for no adjustment). This parameter is mostly used to convert alleles 1 and 2 in a GenePop file to alleles 0 and 1 (with `adjust=-1`) in `simuPOP`. Negative allele (e.g. missing value 0) will be imported as regular allele with module-dependent values (e.g. -1 imported as 255 for standard module).

FSTAT (<http://www2.unil.ch/popgen/softwares/fstat.htm>). This format accepts the following parameters:

adjust Adjust alleles by specified value (default to 0 for no adjustment). This parameter is mostly used to convert alleles 1 and 2 in a GenePop file to alleles 0 and 1 (with `adjust=-1`) in `simuPOP`. Negative allele (e.g. missing value 0) will be imported as regular allele with module-dependent values (e.g. -1 imported as 255 for standard module).

Phylip (Joseph Felsenstein's Phylip format). This function ignores sequence names and import sequences in a haploid (default) or diploid population (if there are even number of sequences). An list of allele names are required to translate symbols to allele names. This format accepts the following parameters:

alleleNames Names of alleles 0, 1, 2, ... as a single string (e.g. 'ACTG') or a list of single-character strings (e.g. ['A', 'C', 'T', 'G']). This will be used to translate symbols into numeric alleles in simuPOP. Allele names will continue to be used as allele names of the returned population.

ploidy Ploidy of the returned population, default to 1 (haploid). There should be even number of sequences if ploidy=2 (haploid) is specified.

MS (output from Richard R. Hudson's MS or msHOT program). The ms program generates npop blocks of nseq haploid chromosomes for command starting with ms nsample nrepeat. By default, the result is imported as a haploid population of size nsample. The population will have nrepeat subpopulations each with the same number of loci but different number of segregating sites. This behavior could be changed by the following parameters:

ploidy If ploidy is set to 2, the sequences will be paired so the population will have nseq/2 individuals. An error will be raised if an odd number of sequences are simulated.

mergeBy By default, replicate samples will be presented as subpopulations. All individuals have the same number of loci but individuals in different subpopulations have different segregating sites. If mergeBy is set to "chrom", the replicates will be presented as separate chromosomes, each with a different set of loci determined by segregating sites.

12.2.10 Function export

simuPOP.utils.**export** (pop, format, *args, **kwargs)
Apply operator `Exporter` to population *pop* in format *format*.

12.3 Module simuPOP.demography

This module provides some commonly used demographic models. In addition to several migration rate generation functions, it provides models that encapsulate complete demographic features of one or more populations (population growth, split, bottleneck, admixture, migration). These models provides:

1. The model itself can be passed to parameter `subPopSize` of a mating scheme to determine the size of the next generation. More importantly, it performs necessary actions of population size change when needed.
2. The model provides attribute `num_gens`, which can be passed to parameter `gens` of `Simulator.evolve` or `Population.evolve` function. A demographic model can also terminate an evolutionary process by returning an empty list so `gens=model.num_gens` is no longer required.

12.3.1 Function migrIslandRates

simuPOP.demography.**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`

12.3.2 Function migrHierarchicalIslandRates

`simuPOP.demography.migrHierarchicalIslandRates (r1, r2, n)`

Return the migration rate matrix for a hierarchical island model where there are different migration rate within and across groups of islands.

r1 Within group migration rates. It can be a number or a list of numbers for each group of the islands.

r2 Across group migration rates which is the probability that someone will migrate to a subpopulation outside of his group. A list of r2 could be specified for each group of the islands.

n Number of islands in each group. E.g. `n=[5, 4]` specifies two groups of islands with 5 and 4 islands each.

For individuals in an island, the probability that it remains in the same island is $1-r1-r2$ ($r1$, $r2$ might vary by island groups), that it migrates to another island in the same group is $r1$ and migrates to another island outside of the group is $r2$. migrate rate to a specific island depends on the size of group.

12.3.3 Function migrSteppingStoneRates

`simuPOP.demography.migrSteppingStoneRates (r, n, circular=False)`

migration rate matrix for 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
```

This function returns `[[1]]` when there **is** only one subpopulation.

12.3.4 Function migrtwoDSteppingStoneRates

`simuPOP.demography.migr2DSteppingStoneRates (r, m, n, diagonal=False, circular=False)`

migration rate matrix for 2D stepping stone model, with or without diagonal neighbors (4 or 8 neighbors for central patches). The boundaries are connected if `circular` is `True`. Otherwise individuals from corner and boundary patches will migrate to their neighbors with higher probability.

12.3.5 class EventBasedModel

`class simuPOP.demography.EventBasedModel`

An event based demographic model in which the demographic changes are triggered by demographic events such as population growth, split, join, and admixture. The population size will be kept constant if no event is applied at a certain generation.

EventBasedModel (`events=[]`, `T=None`, `N0=[]`, `ops=[]`, `infoFields=[]`)

A demographic model that is driven by a list of demographic events. The events should be subclasses of

DemographicEvent, which have similar interface as regular operators with the exception that applicable parameters `begin`, `end`, `step`, `at` are relative to the demographic model, not the population.

plot (*filename*="", *title*="", *initSize*=[])

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter `initSize` for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to `filename`. An optional `title` could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.3.6 class DemographicEvent

class simuPOP.demography.DemographicEvent

A demographic events that will be applied to one or more populations at specified generations. The interface of a DemographicEvent is very similar to an simuPOP operator, but the applicable parameters are handled so that the generations are relative to the demographic model, not the populations to which the event is applied.

DemographicEvent (*ops*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=True, *subPops*=ALL_AVAIL, *infoFields*=[])

Create a demographic event that will be applied at specified generations according to applicability parameters `reps`, `begin`, `end`, `step` and `at`. Parameter `subPops` is usually used to specify the subpopulations affected by the event. One or more simuPOP operators, if specified in `ops`, will be applied when the event happens. Parameters `output` and `infoFields` are currently ignored.

apply (*pop*)

12.3.7 class ExpansionEvent

class simuPOP.demography.ExpansionEvent

A demographic event that increase applicable population size by $N \cdot r$ (to size $N \cdot (1+r)$), or s (to size $N+s$) at each applicable generation. The first model leads to an exponential population expansion model with rate r ($N(t) = N(0) \cdot \exp(r \cdot t)$), where the second model leads to an linear population growth model ($N(t) = N(0) + s \cdot t$) and this is why the parameter is called `slopes`. Note that if both population size and r are small (e.g. $N \cdot r < 1$), the population might not expand as expected.

ExpansionEvent (*rates*=[], *slopes*=[], *capacity*=[], *name*="", *ops*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=True, *subPops*=ALL_AVAIL, *infoFields*=[])

A demographic event that expands all or specified subpopulations (`subPops`) exponentially by a rate of `rates`, or linearly by a slope of `slopes`, unless carry capacity (`capacity`) of the population has been reached. Parameter `rates` can be a single number or a list of rates for all subpopulations. Parameter `slopes` should be a number, or a list of numbers for all subpopulations. `subPops` can be a ALL_AVAIL or a list of subpopulation index or names. `capacity` can be empty (no limit on carrying capacity), or one or more numbers for each of the subpopulations.

apply (*pop*)

12.3.8 class ResizeEvent

class simuPOP.demography.ResizeEvent

A demographic event that resize specified subpopulations

ResizeEvent (*sizes*=[], *names*=[], *removeEmptySubPops*=False, *ops*=[], *output*="", *begin*=0, *end*=-1, *step*=1, *at*=[], *reps*=True, *subPops*=ALL_AVAIL, *infoFields*=[])

A demographic event that resizes given subpopulations `subPops` to new `sizes` (integer type), or `sizes`

proportional to original sizes (if a float number is given). For example, `sizes=[0.5, 500]` will resize the first subpopulation to half of its original size, and the second subpopulation to size 500. If the new size is larger, existing individuals will be copied to sequentially, and repeatedly if needed. If the size of a subpopulation is 0 and `removeEmptySubPops` is `True`, empty subpopulations will be removed. A new set of names could be assigned to the population being resized.

apply (*pop*)

12.3.9 class SplitEvent

class `simuPOP.demography.SplitEvent`

A demographic event that splits a specified population into two or more subpopulations.

SplitEvent (*sizes=[]*, *names=[]*, *ops=[]*, *output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=True*, *subPops=ALL_AVAIL*, *infoFields=[]*)

A demographic event that splits a subpopulation specified by `subPops` to two or more subpopulations, with specified `sizes` and `names`. `sizes` can be a list of numbers, proportions (e.g. `[1., 500]` keeps the original population and copies 500 individuals to create a new subpopulation). Note that `sizes` and `names`, if specified, should include the source subpopulation as its first element.

apply (*pop*)

12.3.10 class MergeEvent

class `simuPOP.demography.MergeEvent`

A demographic event that merges one or more subpopulation to a single one.

MergeEvent (*name=""*, *ops=[]*, *output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=True*, *subPops=ALL_AVAIL*, *infoFields=[]*)

A demographic event that merges subpopulations into a single subpopulation. The merged subpopulation will have the name of the first merged subpopulation unless a separate `name` is supported.

apply (*pop*)

12.3.11 class AdmixtureEvent

class `simuPOP.demography.AdmixtureEvent`

This event implements a population admixture event that mix individuals from specified subpopulations to either a new subpopulation or an existing subpopulation.

AdmixtureEvent (*sizes=[]*, *toSubPop=None*, *name=""*, *ops=[]*, *output=""*, *begin=0*, *end=-1*, *step=1*, *at=[]*, *reps=True*, *subPops=ALL_AVAIL*, *infoFields=[]*)

Create an admixed population by choosing individuals from all or specified subpopulations (`subPops`) and creating an admixed population `toSubPop`. The admixed population will be appended to the population as a new subpopulation with name `name` if `toSubPop` is `None` (default), or replace an existing subpopulation with name or index `toSubPop`. The admixed population consists of individuals from `subPops` according to specified `sizes`. Its size is maximized to have the largest number of individuals from the source population when a new population is created, or equal to the size of the existing destination population. The parameter `sizes` should be a list of float numbers between 0 and 1, and add up to 1 (e.g. `[0.4, 0.4, 0.2]`, although this function ignores the last element and set it to 1 minus the sum of the other numbers). Alternatively, parameter `sizes` can be a list of numbers used to explicitly specify the size of admixed population and number of individuals from each source subpopulation. In all cases, the size of source populations will be kept constant.

apply (*pop*)

12.3.12 class InstantChangeModel

class simuPOP.demography.**InstantChangeModel**

A model for instant population change (growth, resize, merge, split).

InstantChangeModel (*T=None*, *N0=[]*, *G=[]*, *NG=[]*, *ops=[]*, *infoFields=[]*, *removeEmptySubPops=False*)

An instant population growth model that evolves a population from size *N0* to *NT* for *T* generations with population size changes at generation *G* to *NT*. If *G* is a list, multiple population size changes are allowed. In that case, a list (or a nested list) of population size should be provided to parameter *NT*. Both *N0* and *NT* supports fixed (an integer), dynamic (keep passed population size) and proportional (an float number) population size. Optionally, one or more operators (e.g. a migrator) *ops* can be applied to population. Required information fields by these operators should be passed to parameter *infoFields*. If *removeEmpty* option is set to *True*, empty subpopulation will be removed. This option can be used to remove subpopulations.

plot (*filename=""*, *title=""*, *initSize=[]*)

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter *initSize* for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to *filename*. An optional *title* could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.3.13 class ExponentialGrowthModel

class simuPOP.demography.**ExponentialGrowthModel**

A model for exponential population growth with carry capacity

ExponentialGrowthModel (*T=None*, *N0=[]*, *NT=None*, *r=None*, *ops=[]*, *infoFields=[]*)

An exponential population growth model that evolves a population from size *N0* to *NT* for *T* generations with $r \cdot N(t)$ individuals added at each generation. *N0*, *NT* and *r* can be a list of population sizes or growth rates for multiple subpopulations. The initial population will be resized to *N0* (split if necessary). Zero or negative growth rates are allowed. The model will automatically determine *T*, *r* or *NT* if one of them is unspecified. If all of them are specified, *NT* is interpreted as carrying capacity of the model, namely the population will keep constant after it reaches size *NT*. Optionally, one or more operators (e.g. a migrator) *ops* can be applied to population.

plot (*filename=""*, *title=""*, *initSize=[]*)

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter *initSize* for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to *filename*. An optional *title* could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.3.14 class LinearGrowthModel

class simuPOP.demography.**LinearGrowthModel**

A model for linear population growth with carry capacity.

LinearGrowthModel (*T=None*, *N0=[]*, *NT=None*, *r=None*, *ops=[]*, *infoFields=[]*)

An linear population growth model that evolves a population from size *N0* to *NT* for *T* generations with $r \cdot N0$ individuals added at each generation. *N0*, *NT* and *r* can be a list of population sizes or growth rates for multiple subpopulations. The initial population will be resized to *N0* (split if necessary). Zero or

negative growth rates are allowed. The model will automatically determine T , r or NT if one of them is unspecified. If all of them are specified, NT is interpreted as carrying capacity of the model, namely the population will keep constant after it reaches size NT . Optionally, one or more operators (e.g. a migrator) `ops` can be applied to population.

plot (`filename=""`, `title=""`, `initSize=[]`)

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter `initSize` for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to `filename`. An optional `title` could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.3.15 class MultiStageModel

class `simuPOP.demography.MultiStageModel`

A multi-stage demographic model that connects a number of demographic models.

MultiStageModel (`models`, `ops=[]`, `infoFields=[]`)

An multi-stage demographic model that connects specified demographic models `models`. It applies a model to the population until it reaches `num_gens` of the model, or if the model returns `[]`. One or more operators could be specified, which will be applied before a demographic model is applied. Note that the last model will be ignored if it lasts 0 generation.

plot (`filename=""`, `title=""`, `initSize=[]`)

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter `initSize` for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to `filename`. An optional `title` could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.3.16 class OutOfAfricaModel

class `simuPOP.demography.OutOfAfricaModel`

A demographic model for the CHB, CEU, and YRI populations, as defined in Gutenkunst 2009, Plos Genetics. The model is depicted in Figure 2, and the default parameters are listed in Table 1 of this paper.

OutOfAfricaModel (`T0`, `N_A=7300`, `N_AF=12300`, `N_B=2100`, `N_EU0=1000`, `r_EU=0.004`,
`N_AS0=510`, `r_AS=0.0055`, `m_AF_B=0.00025`, `m_AF_EU=3e-05`,
`m_AF_AS=1.9e-05`, `m_EU_AS=9.6e-05`, `T_AF=8800`, `T_B=5600`,
`T_EU_AS=848`, `ops=[]`, `infoFields=[]`, `outcome=['AF', 'EU', 'AS']`, `scale=1`)

Counting **backward in time**, this model evolves a population for `T0` generations (required parameter). The ancient population A started at size `N_A` and expanded at `T_AF` generations from now, to pop AF with size `N_AF`. Pop B split from pop AF at `T_B` generations from now, with size `N_B`; Pop AF remains as `N_AF` individuals. Pop EU and AS split from pop B at `T_EU_AS` generations from now; with size `N_EU0` individuals and `N_AS0` individuals, respectively. Pop EU grew exponentially with rate `r_EU`; Pop AS grew exponentially with rate `r_AS`. The YRI, CEU and CHB samples are drawn from AF, EU and AS populations respectively. Additional operators could be added to `ops`. Information fields required by these operators should be passed to `infoFields`. If a scaling factor `scale` is specified, all population sizes and generation numbers will be divided by a factor of `scale`. This demographic model by default returns all populations (AF, EU, AS) but you can choose to keep only selected subpopulations using parameter `outcome` (e.g. `outcome=['EU', 'AS']`).

This model merges all subpopulations if it is applied to an initial population with multiple subpopulation.

plot (*filename*="", *title*="", *initSize*=[])

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter *initSize* for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to *filename*. An optional *title* could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.3.17 class SettlementOfNewWorldModel

class simuPOP.demography.SettlementOfNewWorldModel

A demographic model for settlement of the new world of Americans, as defined in Gutenkunst 2009, Plos Genetics. The model is depicted in Figure 3, and the default parameters are listed in Table 2 of this paper.

SettlementOfNewWorldModel (*T0*, *N_A*=7300, *N_AF*=12300, *N_B*=2100, *N_EU0*=1500, *r_EU*=0.0023, *N_AS0*=590, *r_AS*=0.0037, *N_MX0*=800, *r_MX*=0.005, *m_AF_B*=0.00025, *m_AF_EU*=3e-05, *m_AF_AS*=1.9e-05, *m_EU_AS*=1.35e-05, *T_AF*=8800, *T_B*=5600, *T_EU_AS*=1056, *T_MX*=864, *f_MX*=0.48, *ops*=[], *infoFields*=[], *outcome*='MXL', *scale*=1)

Counting **backward in time**, this model evolves a population for *T0* generations. The ancient population A started at size *N_A* and expanded at *T_AF* generations from now, to pop AF with size *N_AF*. Pop B split from pop AF at *T_B* generations from now, with size *N_B*; Pop AF remains as *N_AF* individuals. Pop EU and AS split from pop B at *T_EU_AS* generations from now; with size *N_EU0* individuals and *N_AS0* individuals, respectively. Pop EU grew exponentially with final population size *N_EU*; Pop AS grew exponentially with final population size *N_AS*. Pop MX split from pop AS at *T_MX* generations from now with size *N_MX0*, grew exponentially to final size *N_MX*. Migrations are allowed between populations with migration rates *m_AF_B*, *m_EU_AS*, *m_AF_EU*, and *m_AF_AS*. At the end of the evolution, the AF and CHB populations are removed, and the EU and MX populations are merged with *f_MX* proportion for MX. The Mexican American<F19> sample could be sampled from the last single population. Additional operators could be added to *ops*. Information fields required by these operators should be passed to *infoFields*. If a scaling factor *scale* is specified, all population sizes and generation numbers will be divided by a factor of *scale*. This demographic model by default only returns the mixed Mexican America model (*outputcom*='MXL') but you can specify any combination of AF, EU, AS, MX and MXL.

This model merges all subpopulations if it is applied to an initial population with multiple subpopulation.

plot (*filename*="", *title*="", *initSize*=[])

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter *initSize* for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to *filename*. An optional *title* could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.3.18 class CosiModel

class simuPOP.demography.CosiModel

A demographic model for Africa, Asia and Europe, as described in Schaffner et al, Genome Research, 2005, and implemented in the coalescent simulator cosi.

CosiModel (*T0*, *N_A*=12500, *N_AF*=24000, *N_OoA*=7700, *N_AF1*=100000, *N_AS1*=100000, *N_EU1*=100000, *T_AF*=17000, *T_OoA*=3500, *T_EU_AS*=2000, *T_AS_exp*=400, *T_EU_exp*=350, *T_AF_exp*=200, *F_OoA*=0.085, *F_AS*=0.067, *F_EU*=0.02, *F_AF*=0.02, *m_AF_EU*=3.2e-05, *m_AF_AS*=8e-06, *ops*=[], *infoFields*=[], *scale*=1)

Counting **backward in time**, this model evolves a population for a total of T_0 generations. The ancient population *Ancestral* started at size $N_{\text{Ancestral}}$ and expanded at T_{AF} generations from now, to pop *AF* with size N_{AF} . The Out of Africa population split from the *AF* population at T_{OoA} generations ago. The *OoA* population split into two subpopulations *AS* and *EU* but keep the same size. At the generations of $T_{\text{EU_exp}}$, $T_{\text{AS_exp}}$, and $T_{\text{AF_exp}}$ ago, three populations expanded to modern population sizes of N_{AF1} , N_{AS1} and N_{EU1} exponentially, respectively. Migrations are allowed between *AF* and *EU* populations with rate $m_{\text{AF_EU}}$, and between *AF* and *AS* with rate $m_{\text{AF_AS}}$.

Four bottlenecks happens in the *AF*, *OoA*, *EU* and *AS* populations. They are supposed to happen 200 generations after population split and last for 200 generations. The intensity is parameterized in F , which is number of generations divided by twice the effective size during bottleneck. So the bottleneck size is $100/F$.

This model merges all subpopulations if it is applied to a population with multiple subpopulation. Although parameters are configurable, we assume the order of events so dramatically changes of parameters might need to errors. If a scaling factor *scale* is specified, all population sizes and generation numbers will be divided by, and migration rates will be multiplied by a factor of *scale*.

plot (*filename*="", *title*="", *initSize*=[])

Evolve a haploid population using a *RandomSelection* mating scheme using the demographic model. Print population size changes during evolution. An initial population size could be specified using parameter *initSize* for a demographic model with dynamic initial population size. If a filename is specified and if matplotlib is available, this function draws a figure to depict the demographic model and save it to *filename*. An optional *title* could be specified to the figure. Note that this function can not be plot demographic models that works for particular mating schemes (e.g. genotype dependent).

12.4 Module `simuPOP.sampling`

This module provides classes and functions that could be used to draw samples from a `simuPOP` population. These functions accept a list of parameters such as *subPops* ((virtual) subpopulations from which samples will be drawn) and *numOfSamples* (number of samples to draw) and return a list of populations. Both independent individuals and dependent individuals (*Pedigrees*) are supported.

Independent individuals could be drawn from any *Population*. *pedigree* information is not necessary and is usually ignored. Unique IDs are not needed either although such IDs could help you identify samples in the parent *Population*.

Pedigrees could be drawn from multi-generational populations or age-structured populations. All individuals are required to have a unique ID (usually tracked by operator *IdTagger* and are stored in information field *ind_id*). Parents of individuals are usually tracked by operator *PedigreeTagger* and are stored in information fields *father_id* and *mother_id*. If parental information is tracked using operator *ParentsTagger* and information fields *father_idx* and *mother_idx*, a function `sampling.indexToID` can be used to convert index based pedigree to ID based *Pedigree*. Note that *ParentsTagger* can not be used to track *Pedigrees* in age-structured populations because they require parents of each individual resides in a parental generation.

All sampling functions support virtual subpopulations through parameter *subPops*, although sample size specification might vary. This feature allows you to draw samples with specified properties. For example, you could select only female individuals for cases of a female-only disease, or select individuals within certain age-range. If you specify a list of (virtual) subpopulations, you are usually allowed to draw certain number of individuals from each subpopulation.

12.4.1 class `BaseSampler`

class `simuPOP.sampling.BaseSampler`

A sampler extracts individuals from a `simuPOP` population and return them as separate populations. This base

class defines the common interface of all sampling classes, including how samples prepared and returned.

BaseSampler (*subPops=ALL_AVAIL*)

Create a sampler with parameter *subPops*, which will be used to prepare population for sampling. *subPops* should be a list of (virtual) subpopulations from which samples are drawn. The default value is *ALL_AVAIL*, which means all available subpopulations of a Population.

drawSample (*pop*)

Draw and return a sample.

drawSamples (*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

prepareSample (*pop, rearrange*)

Prepare passed population object for sampling according to parameter *subPops*. If samples are drawn from the whole population, a Population will be trimmed if only selected (virtual) subpopulations are used. If samples are drawn separately from specified subpopulations, Population *pop* will be rearranged (if *rearrange==True*) so that each subpopulation corresponds to one element in parameter *subPops*.

12.4.2 class RandomSampler

class `simuPOP.sampling.RandomSampler`

A sampler that draws individuals randomly.

RandomSampler (*sizes, subPops=ALL_AVAIL*)

Creates a random sampler with specified number of individuals.

drawSample (*input_pop*)

Draw a random sample from passed population.

drawSamples (*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

prepareSample (*pop, rearrange*)

Prepare passed population object for sampling according to parameter *subPops*. If samples are drawn from the whole population, a Population will be trimmed if only selected (virtual) subpopulations are used. If samples are drawn separately from specified subpopulations, Population *pop* will be rearranged (if *rearrange==True*) so that each subpopulation corresponds to one element in parameter *subPops*.

12.4.3 Function drawRandomSample

`simuPOP.sampling.drawRandomSample` (*pop, sizes, subPops=ALL_AVAIL*)

Draw *sizes* random individuals from a population. If a single *sizes* is given, individuals are drawn randomly from the whole population or from specified (virtual) subpopulations (parameter *subPops*). Otherwise, a list of numbers should be used to specify number of samples from each subpopulation, which can be all subpopulations if *subPops=ALL_AVAIL* (default), or from each of the specified (virtual) subpopulations. This function returns a population with all extracted individuals.

12.4.4 Function drawRandomSamples

`simuPOP.sampling.drawRandomSamples` (*pop, sizes, numOfSamples=1, subPops=ALL_AVAIL*)

Draw *numOfSamples* random samples from a population and return a list of populations. Please refer to function `drawRandomSample` for more details about parameters *sizes* and *subPops*.

12.4.5 class CaseControlSampler

class `simuPOP.sampling.CaseControlSampler`

A sampler that draws affected and unaffected individuals randomly.

CaseControlSampler (*cases, controls, subPops=ALL_AVAIL*)

Creates a case-control sampler with specified number of cases and controls.

drawSample (*input_pop*)

Draw a case control sample

drawSamples (*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

prepareSample (*input_pop*)

Find out indexes all affected and unaffected individuals.

12.4.6 Function drawCaseControlSample

`simuPOP.sampling.drawCaseControlSample` (*pop, cases, controls, subPops=ALL_AVAIL*)

Draw a case-control samples from a population with *cases* affected and *controls* unaffected individuals. If single *cases* and *controls* are given, individuals are drawn randomly from the whole Population or from specified (virtual) subpopulations (parameter *subPops*). Otherwise, a list of numbers should be used to specify number of cases and controls from each subpopulation, which can be all subpopulations if *subPops=ALL_AVAIL* (default), or from each of the specified (virtual) subpopulations. This function returns a population with all extracted individuals.

12.4.7 Function drawCaseControlSamples

`simuPOP.sampling.drawCaseControlSamples` (*pop, cases, controls, numOfSamples=1, subPops=ALL_AVAIL*)

Draw *numOfSamples* case-control samples from a population with *cases* affected and *controls* unaffected individuals and return a list of populations. Please refer to function `drawCaseControlSample` for a detailed descriptions of parameters.

12.4.8 class PedigreeSampler

class `simuPOP.sampling.PedigreeSampler`

The base class of all pedigree based sampler.

PedigreeSampler (*families, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Creates a pedigree sampler with parameters

families number of families. This can be a number or a list of numbers. In the latter case, specified families are drawn from each subpopulation.

subPops A list of (virtual) subpopulations from which samples are drawn. The default value is *ALL_AVAIL*, which means all available subpopulations of a population.

drawSample (*input_pop*)

Randomly select Pedigrees

drawSamples (*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

family (*id*)

Get the family of individual with *id*.

prepareSample (*pop*, *loci*=[], *infoFields*=[], *ancGens*=True)

Prepare self.pedigree, some pedigree sampler might need additional loci and information fields for this sampler.

12.4.9 class AffectedSibpairSampler

class simuPOP.sampling.AffectedSibpairSampler

A sampler that draws a nuclear family with two affected offspring.

AffectedSibpairSampler (*families*, *subPops*=ALL_AVAIL, *idField*='ind_id', *fatherField*='father_id', *motherField*='mother_id')

Initialize an affected sibpair sampler.

drawSample (*input_pop*)

Randomly select Pedigrees

drawSamples (*pop*, *numOfSamples*)

Draw multiple samples and return a list of populations.

family (*id*)

Return *id*, its spouse and their children

prepareSample (*input_pop*)

Find the father or all affected sibpair families

12.4.10 Function drawAffectedSibpairSample

simuPOP.sampling.drawAffectedSibpairSample (*pop*, *families*, *subPops*=ALL_AVAIL, *idField*='ind_id', *fatherField*='father_id', *motherField*='mother_id')

Draw affected sibpair samples from a population. If a single *families* is given, affected sibpairs and their parents are drawn randomly from the whole population or from specified (virtual) subpopulations (parameter *subPops*). Otherwise, a list of numbers should be used to specify number of families from each subpopulation, which can be all subpopulations if *subPops*=ALL_AVAIL (default), or from each of the specified (virtual) subpopulations. This function returns a population that contains extracted individuals.

12.4.11 Function drawAffectedSibpairSamples

simuPOP.sampling.drawAffectedSibpairSamples (*pop*, *families*, *numOfSamples*=1, *subPops*=ALL_AVAIL, *idField*='ind_id', *fatherField*='father_id', *motherField*='mother_id')

Draw *numOfSamples* affected sibpair samples from population *pop* and return a list of populations. Please refer to function drawAffectedSibpairSample for a description of other parameters.

12.4.12 class NuclearFamilySampler

class simuPOP.sampling.NuclearFamilySampler

A sampler that draws nuclear families with specified number of affected parents and offspring.

NuclearFamilySampler (*families, numOffspring, affectedParents=0, affectedOffspring=0, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Creates a nuclear family sampler with parameters

families number of families. This can be a number or a list of numbers. In the latter case, specified families are drawn from each subpopulation.

numOffspring number of offspring. This can be a fixed number or a range [min, max].

affectedParents number of affected parents. This can be a fixed number or a range [min, max].

affectedOffspring number of affected offspring. This can be a fixed number of a range [min, max].

subPops A list of (virtual) subpopulations from which samples are drawn. The default value is ALL_AVAIL, which means all available subpopulations of a population.

drawSample (*input_pop*)

Randomly select Pedigrees

drawSamples (*pop, numOfSamples*)

Draw multiple samples and return a list of populations.

family (*id*)

Return id, its spouse and their children

prepareSample (*input_pop*)

Prepare self.pedigree, some pedigree sampler might need additional loci and information fields for this sampler.

12.4.13 Function drawNuclearFamilySample

`simuPOP.sampling.drawNuclearFamilySample` (*pop, families, numOffspring, affectedParents=0, affectedOffspring=0, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Draw nuclear families from a population. Number of offspring, number of affected parents and number of affected offspring should be specified using parameters `numOffspring`, `affectedParents` and `affectedOffspring`, which can all be a single number, or a range [*a*, *b*] (*b* is included). If a single `families` is given, Pedigrees are drawn randomly from the whole population or from specified (virtual) subpopulations (parameter `subPops`). Otherwise, a list of numbers should be used to specify numbers of families from each subpopulation, which can be all subpopulations if `subPops=ALL_AVAIL` (default), or from each of the specified (virtual) subpopulations. This function returns a population that contains extracted individuals.

12.4.14 Function drawNuclearFamilySamples

`simuPOP.sampling.drawNuclearFamilySamples` (*pop, families, numOffspring, affectedParents=0, affectedOffspring=0, numOfSamples=1, subPops=ALL_AVAIL, idField='ind_id', fatherField='father_id', motherField='mother_id'*)

Draw `numOfSamples` affected sibpair samples from population `pop` and return a list of populations. Please refer to function `drawNuclearFamilySample` for a description of other parameters.

12.4.15 class ThreeGenFamilySampler

class `simuPOP.sampling.ThreeGenFamilySampler`

A sampler that draws three-generation families with specified pedigree size and number of affected individuals.

ThreeGenFamilySampler (*families*, *numOffspring*, *pedSize*, *numOfAffected=0*, *subPops=ALL_AVAIL*, *idField='ind_id'*, *fatherField='father_id'*, *motherField='mother_id'*)

families number of families. This can be a number or a list of numbers. In the latter case, specified families are drawn from each subpopulation.

numOffspring number of offspring. This can be a fixed number or a range [min, max].

pedSize number of individuals in the Pedigree. This can be a fixed number or a range [min, max].

numAffected number of affected individuals in the Pedigree. This can be a fixed number or a range [min, max]

subPops A list of (virtual) subpopulations from which samples are drawn. The default value is ALL_AVAIL, which means all available subpopulations of a population.

drawSample (*input_pop*)
Randomly select Pedigrees

drawSamples (*pop*, *numOfSamples*)
Draw multiple samples and return a list of populations.

family (*id*)
Return id, its spouse, their children, children's spouse and grandchildren

prepareSample (*input_pop*)
Prepare self.pedigree, some pedigree sampler might need additional loci and information fields for this sampler.

12.4.16 Function drawThreeGenFamilySample

`simuPOP.sampling.drawThreeGenFamilySample` (*pop*, *families*, *numOffspring*, *pedSize*, *numOfAffected=0*, *subPops=ALL_AVAIL*, *idField='ind_id'*, *fatherField='father_id'*, *motherField='mother_id'*)

Draw three-generation families from a population. Such families consist of grant parents, their children, spouse of these children, and grand children. Number of offspring, total number of individuals, and total number of affected individuals in a pedigree should be specified using parameters *numOffspring*, *pedSize* and *numOfAffected*, which can all be a single number, or a range [a, b] (b is included). If a single *families* is given, Pedigrees are drawn randomly from the whole Population or from specified (virtual) subpopulations (parameter *subPops*). Otherwise, a list of numbers should be used to specify numbers of families from each subpopulation, which can be all subpopulations if *subPops=ALL_AVAIL* (default), or from each of the specified (virtual) subpopulations. This function returns a population that contains extracted individuals.

12.4.17 Function drawThreeGenFamilySamples

`simuPOP.sampling.drawThreeGenFamilySamples` (*pop*, *families*, *numOffspring*, *pedSize*, *numOfAffected=0*, *numOfSamples=1*, *subPops=ALL_AVAIL*, *idField='ind_id'*, *fatherField='father_id'*, *motherField='mother_id'*)

Draw *numOfSamples* three-generation pedigree samples from population *pop* and return a list of populations. Please refer to function `drawThreeGenFamilySample` for a description of other parameters.

12.4.18 class CombinedSampler

class `simuPOP.sampling.CombinedSampler`

A combined sampler accepts a list of sampler objects, draw samples and combine the returned sample into a single population. An `idField` is required to use this sampler, which will be used to remove extra copies of individuals who have been drawn by different samplers.

CombinedSampler (*samplers*=[], *idField*='ind_id')

samplers A list of samplers

drawSample (*pop*)

Draw and return a sample.

drawSamples (*pop*, *numOfSamples*)

Draw multiple samples and return a list of populations.

prepareSample (*pop*, *rearrange*)

Prepare passed population object for sampling according to parameter `subPops`. If samples are drawn from the whole population, a Population will be trimmed if only selected (virtual) subpopulations are used. If samples are drawn separately from specified subpopulations, Population `pop` will be rearranged (if `rearrange==True`) so that each subpopulation corresponds to one element in parameter `subPops`.

12.4.19 Function drawCombinedSample

`simuPOP.sampling.drawCombinedSample` (*pop*, *samplers*, *idField*='ind_id')

Draw different types of samples using a list of `samplers`. A Population consists of all individuals from these samples will be returned. An `idField` that stores a unique ID for all individuals is needed to remove duplicated individuals who are drawn multiple `numOfSamples` from these samplers.

12.4.20 Function drawCombinedSamples

`simuPOP.sampling.drawCombinedSamples` (*pop*, *samplers*, *numOfSamples*=1, *idField*='ind_id')

Draw combined samples `numOfSamples` `numOfSamples` and return a list of populations. Please refer to function `drawCombinedSample` for details about parameters `samplers` and `idField`.

12.5 Module `simuPOP.gsl`

This module exposes the following GSL (GUN Scientific Library) functions used by `simuPOP` to the user interface. Although more functions may be added from time to time, this module is not intended to become a complete wrapper for GSL. Please refer to the GSL reference manual (http://www.gnu.org/software/gsl/manual/html_node/) for details about these functions. Note that random number generation functions are wrapped into the `simuPOP.RNG` class.

- `gsl_cdf_gaussian_P(x, sigma)`
- `gsl_cdf_gaussian_Q(x, sigma)`
- `gsl_cdf_gaussian_Pinv(P, sigma)`
- `gsl_cdf_gaussian_Qinv(Q, sigma)`
- `gsl_cdf_ugaussian_P(x)`
- `gsl_cdf_ugaussian_Q(x)`
- `gsl_cdf_ugaussian_Pinv(P)`

- `gsl_cdf_ugaussian_Qinv(Q)`
- `gsl_cdf_exponential_P(x, mu)`
- `gsl_cdf_exponential_Q(x, mu)`
- `gsl_cdf_exponential_Pinv(P, mu)`
- `gsl_cdf_exponential_Qinv(Q, mu)`
- `gsl_cdf_chisq_P(x, nu)`
- `gsl_cdf_chisq_Q(x, nu)`
- `gsl_cdf_chisq_Pinv(P, nu)`
- `gsl_cdf_chisq_Qinv(Q, nu)`
- `gsl_cdf_gamma_P(x, a, b)`
- `gsl_cdf_gamma_Q(x, a, b)`
- `gsl_cdf_gamma_Pinv(P, a, b)`
- `gsl_cdf_gamma_Qinv(Q, a, b)`
- `gsl_ran_gamma_pdf(x, a, b)`
- `gsl_cdf_beta_P(x, a, b)`
- `gsl_cdf_beta_Q(x, a, b)`
- `gsl_cdf_beta_Pinv(P, a, b)`
- `gsl_cdf_beta_Qinv(Q, a, b)`
- `gsl_ran_beta_pdf(x, a, b)`
- `gsl_cdf_binomial_P(k, p, n)`
- `gsl_cdf_binomial_Q(k, p, n)`
- `gsl_ran_binomial_pdf(k, p, n)`
- `gsl_cdf_poisson_P(k, mu)`
- `gsl_cdf_poisson_Q(k, mu)`
- `gsl_ran_poisson_pdf(k, mu)`

S

- `simuOpt`, [337](#)
- `simuPOP.demography`, [347](#)
- `simuPOP.gsl`, [360](#)
- `simuPOP.sampling`, [354](#)
- `simuPOP.utils`, [338](#)

Symbols

`__cmp__()` (Individual method), 264
`__cmp__()` (Population method), 267
`__cmp__()` (Simulator method), 275

A

`absIndIndex()` (Population method), 266
`absLocusIndex()` (GenoStruTrait method), 261
`acgtMutate()` (built-in function), 333
`AcgtMutator` (built-in class), 311
`AcgtMutator()` (AcgtMutator method), 311
`add()` (Simulator method), 275
`addChrom()` (Population method), 266
`addChromFrom()` (Population method), 267
`addIndFrom()` (Population method), 267
`addInfoFields()` (Population method), 267
`addLoci()` (Population method), 267
`addLociFrom()` (Population method), 267
`AdmixtureEvent` (class in `simuPOP.demography`), 350
`AdmixtureEvent()` (`simuPOP.demography.AdmixtureEvent` method), 350
`affected()` (Individual method), 263
`AffectedSibpairSampler` (class in `simuPOP.sampling`), 357
`AffectedSibpairSampler()` (`simuPOP.sampling.AffectedSibpairSampler` method), 357
`AffectionSplitter` (built-in class), 277
`AffectionSplitter()` (`AffectionSplitter` method), 277
`allele()` (Individual method), 263
`alleleChar()` (Individual method), 264
`alleleLineage()` (Individual method), 264
`alleleName()` (GenoStruTrait method), 261
`alleleNames()` (GenoStruTrait method), 262
`allIndividuals()` (Population method), 272
`ancestor()` (Population method), 267
`ancestralGens()` (Population method), 267
`apply()` (BaseOperator method), 296
`apply()` (BasePenetrance method), 312

`apply()` (BaseQuanTrait method), 314
`apply()` (`simuPOP.demography.AdmixtureEvent` method), 350
`apply()` (`simuPOP.demography.DemographicEvent` method), 349
`apply()` (`simuPOP.demography.ExpansionEvent` method), 349
`apply()` (`simuPOP.demography.MergeEvent` method), 350
`apply()` (`simuPOP.demography.ResizeEvent` method), 350
`apply()` (`simuPOP.demography.SplitEvent` method), 350
`applyToIndividual()` (BasePenetrance method), 312
`asPedigree()` (Population method), 272
`asPopulation()` (Pedigree method), 274

B

`backwardMigrate()` (built-in function), 334
`BackwardMigrator` (built-in class), 301
`BackwardMigrator()` (`BackwardMigrator` method), 301
`BaseMutator` (built-in class), 307
`BaseMutator()` (`BaseMutator` method), 307
`BaseOperator` (built-in class), 295
`BaseOperator()` (`BaseOperator` method), 296
`BasePenetrance` (built-in class), 311
`BasePenetrance()` (`BasePenetrance` method), 312
`BaseQuanTrait` (built-in class), 314
`BaseQuanTrait()` (`BaseQuanTrait` method), 314
`BaseSampler` (class in `simuPOP.sampling`), 354
`BaseSampler()` (`simuPOP.sampling.BaseSampler` method), 355
`BaseSelector` (built-in class), 315
`BaseSelector()` (`BaseSelector` method), 316
`BaseVspSplitter` (built-in class), 276
`BaseVspSplitter()` (`BaseVspSplitter` method), 276

C

`CaseControlSampler` (class in `simuPOP.sampling`), 356
`CaseControlSampler()` (`simuPOP.sampling.CaseControlSampler` method), 356
`chooseParents()` (`CombinedParentsChooser` method), 284

- chooseParents() (PolyParentsChooser method), 283
 - chooseParents() (PyParentsChooser method), 284
 - chooseParents() (RandomParentChooser method), 283
 - chooseParents() (RandomParentsChooser method), 283
 - chooseParents() (SequentialParentChooser method), 282
 - chromBegin() (GenoStruTrait method), 262
 - chromByName() (GenoStruTrait method), 262
 - chromEnd() (GenoStruTrait method), 262
 - chromLocusPair() (GenoStruTrait method), 262
 - chromName() (GenoStruTrait method), 262
 - chromNames() (GenoStruTrait method), 262
 - chromType() (GenoStruTrait method), 262
 - chromTypes() (GenoStruTrait method), 262
 - clearChromosome() (GenoTransmitter method), 303
 - clone() (BaseOperator method), 296
 - clone() (BaseVspSplitter method), 276
 - clone() (Pedigree method), 273
 - clone() (Population method), 267
 - clone() (Simulator method), 275
 - CloneGenoTransmitter (built-in class), 304
 - CloneGenoTransmitter() (CloneGenoTransmitter method), 304
 - CloneMating (built-in class), 286
 - CloneMating() (CloneMating method), 286
 - closeOutput() (built-in function), 291
 - CombinedParentsChooser (built-in class), 284
 - CombinedParentsChooser() (CombinedParentsChooser method), 284
 - CombinedSampler (class in simuPOP.sampling), 360
 - CombinedSampler() (simuPOP.sampling.CombinedSampler method), 360
 - CombinedSplitter (built-in class), 279
 - CombinedSplitter() (CombinedSplitter method), 279
 - ConditionalMating (built-in class), 281
 - ConditionalMating() (ConditionalMating method), 281
 - contextMutate() (built-in function), 333
 - ContextMutator (built-in class), 310
 - ContextMutator() (ContextMutator method), 310
 - ControlledOffspringGenerator (built-in class), 286
 - ControlledOffspringGenerator() (ControlledOffspringGenerator method), 286
 - ControlledRandomMating (built-in class), 288
 - ControlledRandomMating() (ControlledRandomMating method), 288
 - copyChromosome() (GenoTransmitter method), 303
 - copyChromosomes() (GenoTransmitter method), 303
 - CosiModel (class in simuPOP.demography), 353
 - CosiModel() (simuPOP.demography.CosiModel method), 353
- ## D
- DemographicEvent (class in simuPOP.demography), 349
 - DemographicEvent() (simuPOP.demography.DemographicEvent method), 349
 - describeEvolProcess() (built-in function), 291
 - DiscardIf (built-in class), 330
 - discardIf() (built-in function), 333
 - DiscardIf() (DiscardIf method), 330
 - done() (simuPOP.utils.ProgressBar method), 342
 - draw() (WeightedSampler method), 290
 - drawAffectedSibpairSample() (in module simuPOP.sampling), 357
 - drawAffectedSibpairSamples() (in module simuPOP.sampling), 357
 - drawCaseControlSample() (in module simuPOP.sampling), 356
 - drawCaseControlSamples() (in module simuPOP.sampling), 356
 - drawCombinedSample() (in module simuPOP.sampling), 360
 - drawCombinedSamples() (in module simuPOP.sampling), 360
 - drawNuclearFamilySample() (in module simuPOP.sampling), 358
 - drawNuclearFamilySamples() (in module simuPOP.sampling), 358
 - drawRandomSample() (in module simuPOP.sampling), 355
 - drawRandomSamples() (in module simuPOP.sampling), 355
 - drawSample() (simuPOP.sampling.AffectedSibpairSampler method), 357
 - drawSample() (simuPOP.sampling.BaseSampler method), 355
 - drawSample() (simuPOP.sampling.CaseControlSampler method), 356
 - drawSample() (simuPOP.sampling.CombinedSampler method), 360
 - drawSample() (simuPOP.sampling.NuclearFamilySampler method), 358
 - drawSample() (simuPOP.sampling.PedigreeSampler method), 356
 - drawSample() (simuPOP.sampling.RandomSampler method), 355
 - drawSample() (simuPOP.sampling.ThreeGenFamilySampler method), 359
 - drawSamples() (simuPOP.sampling.AffectedSibpairSampler method), 357
 - drawSamples() (simuPOP.sampling.BaseSampler method), 355
 - drawSamples() (simuPOP.sampling.CaseControlSampler method), 356
 - drawSamples() (simuPOP.sampling.CombinedSampler method), 360
 - drawSamples() (simuPOP.sampling.NuclearFamilySampler method), 358
 - drawSamples() (simuPOP.sampling.PedigreeSampler method), 356

- drawSamples() (simuPOP.sampling.RandomSampler method), 355
- drawSamples() (simuPOP.sampling.ThreeGenFamilySampler method), 359
- drawSamples() (WeightedSampler method), 290
- drawThreeGenFamilySample() (in module simuPOP.sampling), 359
- drawThreeGenFamilySamples() (in module simuPOP.sampling), 359
- dump() (built-in function), 333
- Dumper (built-in class), 331
- Dumper() (Dumper method), 331
- dvars() (Population method), 267
- dvars() (Simulator method), 275
- ## E
- EventBasedModel (class in simuPOP.demography), 348
- EventBasedModel() (simuPOP.demography.EventBasedModel method), 348
- evolve() (Population method), 272
- evolve() (Simulator method), 275
- ExpansionEvent (class in simuPOP.demography), 349
- ExpansionEvent() (simuPOP.demography.ExpansionEvent method), 349
- ExponentialGrowthModel (class in simuPOP.demography), 351
- ExponentialGrowthModel() (simuPOP.demography.ExponentialGrowthModel method), 351
- export() (in module simuPOP.utils), 347
- Exporter (class in simuPOP.utils), 343
- Exporter() (simuPOP.utils.Exporter method), 346
- extract() (Simulator method), 276
- extractIndividuals() (Population method), 267
- extractSubPops() (Population method), 268
- ## F
- family() (simuPOP.sampling.AffectedSibpairSampler method), 357
- family() (simuPOP.sampling.NuclearFamilySampler method), 358
- family() (simuPOP.sampling.PedigreeSampler method), 356
- family() (simuPOP.sampling.ThreeGenFamilySampler method), 359
- freq() (simuPOP.utils.Trajectory method), 339
- func() (simuPOP.utils.Trajectory method), 339
- function
- loadPopulation, 46
- ## G
- GenoStruTrait
- chromName, 23
 - chromType, 23
 - infoField, 23
 - infoFields, 23
 - locusPos, 23
 - numChrom, 23
 - numLoci, 23
 - ploidy, 23
 - ploidyName, 23
- GenoStruTrait (built-in class), 261
- GenoStruTrait() (GenoStruTrait method), 261
- GenoTransmitter (built-in class), 303
- GenoTransmitter() (GenoTransmitter method), 303
- genotype() (Individual method), 264
- genotype() (Population method), 268
- GenotypeSplitter (built-in class), 278
- GenotypeSplitter() (GenotypeSplitter method), 278
- genotypic structure, 23
- getRNG() (built-in function), 292
- ## H
- HaplodiploidGenoTransmitter (built-in class), 305
- HaplodiploidGenoTransmitter() (HaplodiploidGenoTransmitter method), 305
- HaplodiploidMating (built-in class), 287
- HaplodiploidMating() (HaplodiploidMating method), 288
- HermaphroditicMating (built-in class), 288
- HermaphroditicMating() (HermaphroditicMating method), 288
- HeteroMating (built-in class), 280
- HeteroMating() (HeteroMating method), 280
- HomoMating (built-in class), 280
- HomoMating() (HomoMating method), 280
- ## I
- identifyAncestors() (Pedigree method), 273
- identifyFamilies() (Pedigree method), 273
- identifyOffspring() (Pedigree method), 273
- IdTagger (built-in class), 318
- IdTagger() (IdTagger method), 318
- IfElse (built-in class), 329
- IfElse() (IfElse method), 329
- importPopulation() (in module simuPOP.utils), 346
- indByID() (Pedigree method), 273
- indByID() (Population method), 268
- index
- absolute, 12
 - relative, 12
- indexesOfLoci() (GenoStruTrait method), 262
- indInfo() (Population method), 268
- Individual, 261
- Individual (built-in class), 263
- Individual() (Individual method), 263
- individual() (Population method), 268
- individuals() (Population method), 268

individualsWithRelatives() (Pedigree method), 273
 info() (Individual method), 264
 InfoEval (built-in class), 299
 infoEval() (built-in function), 333
 InfoEval() (InfoEval method), 299
 InfoExec (built-in class), 299
 infoExec() (built-in function), 333
 InfoExec() (InfoExec method), 300
 infoField() (GenoStruTrait method), 262
 infoFields() (GenoStruTrait method), 262
 infoIdx() (GenoStruTrait method), 262
 InfoSplitter (built-in class), 277
 InfoSplitter() (InfoSplitter method), 277
 InheritTagger (built-in class), 319
 InheritTagger() (InheritTagger method), 319
 InitGenotype (built-in class), 297
 initGenotype() (built-in function), 333
 InitGenotype() (InitGenotype method), 297
 initialize() (CombinedParentsChooser method), 284
 initialize() (PolyParentsChooser method), 283
 initialize() (PyParentsChooser method), 284
 initialize() (RandomParentChooser method), 283
 initialize() (RandomParentsChooser method), 283
 initialize() (SequentialParentChooser method), 282
 initializer, 296
 InitInfo (built-in class), 297
 initInfo() (built-in function), 334
 InitInfo() (InitInfo method), 297
 InitLineage (built-in class), 298
 InitLineage() (InitLineage method), 298
 InitSex (built-in class), 296
 initSex() (built-in function), 334
 InitSex() (InitSex method), 296
 InstantChangeModel (class in simuPOP.demography), 351
 InstantChangeModel() (simuPOP.demography.InstantChangeModel method), 351

K

kAlleleMutate() (built-in function), 334
 KAlleleMutator (built-in class), 308
 KAlleleMutator() (KAlleleMutator method), 308

L

lineage() (Individual method), 264
 lineage() (Population method), 268
 LinearGrowthModel (class in simuPOP.demography), 351
 LinearGrowthModel() (simuPOP.demography.LinearGrowthModel method), 351
 loadPedigree() (built-in function), 291
 loadPopulation() (built-in function), 291
 locateRelatives() (Pedigree method), 273
 lociByNames() (GenoStruTrait method), 262

lociDist() (GenoStruTrait method), 262
 lociNames() (GenoStruTrait method), 262
 lociPos() (GenoStruTrait method), 262
 locusByName() (GenoStruTrait method), 263
 locusName() (GenoStruTrait method), 263
 locusPos() (GenoStruTrait method), 263

M

MaPenetrance (built-in class), 312
 maPenetrance() (built-in function), 334
 MaPenetrance() (MaPenetrance method), 312
 MapPenetrance (built-in class), 312
 mapPenetrance() (built-in function), 334
 MapPenetrance() (MapPenetrance method), 312
 MapSelector (built-in class), 316
 MapSelector() (MapSelector method), 316
 MaSelector (built-in class), 316
 MaSelector() (MaSelector method), 316
 mating scheme, 157, 280
 MatingScheme (built-in class), 280
 MatingScheme() (MatingScheme method), 280
 matrixMutate() (built-in function), 334
 MatrixMutator (built-in class), 308
 MatrixMutator() (MatrixMutator method), 308
 MendelianGenoTransmitter (built-in class), 304
 MendelianGenoTransmitter() (MendelianGenoTransmitter method), 304
 MergeEvent (class in simuPOP.demography), 350
 MergeEvent() (simuPOP.demography.MergeEvent method), 350
 MergeSubPops (built-in class), 302
 mergeSubPops() (built-in function), 334
 MergeSubPops() (MergeSubPops method), 302
 mergeSubPops() (Population method), 268
 migr2DSteppingStoneRates() (in module simuPOP.demography), 348
 migrate() (built-in function), 334
 Migrator, 300
 Migrator (built-in class), 300
 Migrator() (Migrator method), 301
 migrHierarchicalIslandRates() (in module simuPOP.demography), 348
 migrIslandRates() (in module simuPOP.demography), 347
 migrSteppingStoneRates() (in module simuPOP.demography), 348
 MitochondrialGenoTransmitter (built-in class), 305
 MitochondrialGenoTransmitter() (MitochondrialGenoTransmitter method), 305
 mixedMutate() (built-in function), 335
 MixedMutator (built-in class), 310
 MixedMutator() (MixedMutator method), 310
 MIPenetrance (built-in class), 313
 mlPenetrance() (built-in function), 335

MIPenetrance() (MIPenetrance method), 313
 MISelector (built-in class), 317
 MISelector() (MISelector method), 317
 moduleInfo, 21
 moduleInfo() (built-in function), 292
 MonogamousMating (built-in class), 287
 MonogamousMating() (MonogamousMating method), 287
 MultiStageModel (class in simuPOP.demography), 352
 MultiStageModel() (simuPOP.demography.MultiStageModel method), 352
 mutants() (Individual method), 264
 mutants() (Population method), 269
 mutants() (simuPOP.utils.Trajectory method), 339
 Mutation, 307
 mutators() (simuPOP.utils.Trajectory method), 339

N

name() (AffectionSplitter method), 277
 name() (BaseVspSplitter method), 276
 name() (CombinedSplitter method), 279
 name() (GenotypeSplitter method), 279
 name() (InfoSplitter method), 277
 name() (ProductSplitter method), 279
 name() (ProportionSplitter method), 278
 name() (RangeSplitter method), 278
 name() (RNG method), 289
 name() (SexSplitter method), 277
 NoneOp (built-in class), 331
 NoneOp() (NoneOp method), 331
 NuclearFamilySampler (class in simuPOP.sampling), 357
 NuclearFamilySampler() (simuPOP.sampling.NuclearFamilySampler method), 357
 numChrom() (GenoStruTrait method), 263
 numLoci() (GenoStruTrait method), 263
 numRep() (Simulator method), 276
 numSubPop() (Population method), 269
 numVirtualSubPop() (AffectionSplitter method), 277
 numVirtualSubPop() (BaseVspSplitter method), 276
 numVirtualSubPop() (CombinedSplitter method), 279
 numVirtualSubPop() (GenotypeSplitter method), 279
 numVirtualSubPop() (InfoSplitter method), 277
 numVirtualSubPop() (Population method), 269
 numVirtualSubPop() (ProductSplitter method), 279
 numVirtualSubPop() (ProportionSplitter method), 278
 numVirtualSubPop() (RangeSplitter method), 278
 numVirtualSubPop() (SexSplitter method), 277

O

OffspringGenerator (built-in class), 284
 OffspringGenerator() (OffspringGenerator method), 284
 OffspringTagger (built-in class), 320
 OffspringTagger() (OffspringTagger method), 320

operator
 Stat, 45
 OutOfAfricaModel (class in simuPOP.demography), 352
 OutOfAfricaModel() (simuPOP.demography.OutOfAfricaModel method), 352

P

parallelizable() (PedigreeMating method), 282
 ParentsTagger (built-in class), 320
 ParentsTagger() (ParentsTagger method), 320
 Pause (built-in class), 332
 Pause() (Pause method), 332
 Pedigree, 261
 Pedigree (built-in class), 272
 Pedigree() (Pedigree method), 272
 PedigreeMating (built-in class), 281
 PedigreeMating() (PedigreeMating method), 282
 PedigreeSampler (class in simuPOP.sampling), 356
 PedigreeSampler() (simuPOP.sampling.PedigreeSampler method), 356
 PedigreeTagger (built-in class), 320
 PedigreeTagger() (PedigreeTagger method), 320
 penetrance, 311
 ploidy() (GenoStruTrait method), 263
 ploidyName() (GenoStruTrait method), 263
 plot() (simuPOP.demography.CosiModel method), 354
 plot() (simuPOP.demography.EventBasedModel method), 349
 plot() (simuPOP.demography.ExponentialGrowthModel method), 351
 plot() (simuPOP.demography.InstantChangeModel method), 351
 plot() (simuPOP.demography.LinearGrowthModel method), 352
 plot() (simuPOP.demography.MultiStageModel method), 352
 plot() (simuPOP.demography.OutOfAfricaModel method), 352
 plot() (simuPOP.demography.SettlementOfNewWorldModel method), 353
 pointMutate() (built-in function), 335
 PointMutator (built-in class), 310
 PointMutator() (PointMutator method), 310
 PolygamousMating (built-in class), 287
 PolygamousMating() (PolygamousMating method), 287
 PolyParentsChooser (built-in class), 283
 PolyParentsChooser() (PolyParentsChooser method), 283
 popSize() (Population method), 269
 Population, 32, 261
 Population, 45
 save, 46
 vars, 45
 Population (built-in class), 265
 Population() (Population method), 266

- ul style="list-style-type: none; padding-left: 0;">
- population() (Simulator method), 276
- populations() (Simulator method), 276
- prepareSample() (simuPOP.sampling.AffectedSibpairSampler method), 357
- prepareSample() (simuPOP.sampling.BaseSampler method), 355
- prepareSample() (simuPOP.sampling.CaseControlSampler method), 356
- prepareSample() (simuPOP.sampling.CombinedSampler method), 360
- prepareSample() (simuPOP.sampling.NuclearFamilySampler method), 358
- prepareSample() (simuPOP.sampling.PedigreeSampler method), 357
- prepareSample() (simuPOP.sampling.RandomSampler method), 355
- prepareSample() (simuPOP.sampling.ThreeGenFamilySampler method), 359
- ProductSplitter (built-in class), 279
- ProductSplitter() (ProductSplitter method), 279
- ProgressBar (class in simuPOP.utils), 341
- ProgressBar() (simuPOP.utils.ProgressBar method), 341
- ProportionSplitter (built-in class), 278
- ProportionSplitter() (ProportionSplitter method), 278
- push() (Population method), 269
- PyEval (built-in class), 298
- pyEval() (built-in function), 335
- PyEval() (PyEval method), 298
- PyExec (built-in class), 299
- pyExec() (built-in function), 335
- PyExec() (PyExec method), 299
- PyMIPenetrance (built-in class), 314
- pyMIPenetrance() (built-in function), 335
- PyMIPenetrance() (PyMIPenetrance method), 314
- PyMISelector (built-in class), 318
- PyMISelector() (PyMISelector method), 318
- pyMutate() (built-in function), 335
- PyMutator (built-in class), 309
- PyMutator() (PyMutator method), 309
- PyOperator (built-in class), 330
- PyOperator() (PyOperator method), 331
- PyOutput (built-in class), 298
- PyOutput() (PyOutput method), 298
- PyParentsChooser (built-in class), 284
- PyParentsChooser() (PyParentsChooser method), 284
- PyPenetrance (built-in class), 313
- pyPenetrance() (built-in function), 335
- PyPenetrance() (PyPenetrance method), 313
- PyQuanTrait (built-in class), 315
- pyQuanTrait() (built-in function), 336
- PyQuanTrait() (PyQuanTrait method), 315
- PySelector (built-in class), 317
- PySelector() (PySelector method), 318
- PyTagger (built-in class), 321
- PyTagger() (PyTagger method), 321
- Q**
- quantitative trait, 314
- R**
- r, 17
- randBinomial() (RNG method), 289
- randChisq() (RNG method), 289
- randExponential() (RNG method), 290
- randGamma() (RNG method), 290
- randGeometric() (RNG method), 290
- randInt() (RNG method), 290
- randMultinomial() (RNG method), 290
- randNormal() (RNG method), 290
- RandomMating (built-in class), 287
- RandomMating() (RandomMating method), 287
- RandomParentChooser (built-in class), 282
- RandomParentChooser() (RandomParentChooser method), 283
- RandomParentsChooser (built-in class), 283
- RandomParentsChooser() (RandomParentsChooser method), 283
- RandomSampler (class in simuPOP.sampling), 355
- RandomSampler() (simuPOP.sampling.RandomSampler method), 355
- RandomSelection (built-in class), 286
- RandomSelection() (RandomSelection method), 286
- randPoisson() (RNG method), 290
- randTruncatedBinomial() (RNG method), 290
- randTruncatedPoisson() (RNG method), 290
- randUniform() (RNG method), 290
- RangeSplitter (built-in class), 278
- RangeSplitter() (RangeSplitter method), 278
- recodeAlleles() (Population method), 269
- Recombinator (built-in class), 305
- Recombinator() (Recombinator method), 306
- removeIndividuals() (Population method), 269
- removeInfoFields() (Population method), 270
- removeLoci() (Population method), 270
- removeSubPops() (Population method), 270
- reset() (IdTagger method), 319
- resize() (Population method), 270
- ResizeEvent (class in simuPOP.demography), 349
- ResizeEvent() (simuPOP.demography.ResizeEvent method), 349
- ResizeSubPops (built-in class), 303
- resizeSubPops() (built-in function), 336
- ResizeSubPops() (ResizeSubPops method), 303
- RNG (built-in class), 289
- RNG() (RNG method), 289
- S**
- save() (Pedigree method), 274

- save() (Population method), 270
 saveCSV() (in module simuPOP.utils), 342
 SavePopulation (built-in class), 332
 SavePopulation() (SavePopulation method), 332
 seed() (RNG method), 290
 selection, 315
 SelfingGenoTransmitter (built-in class), 304
 SelfingGenoTransmitter() (SelfingGenoTransmitter method), 304
 SelfMating (built-in class), 288
 SelfMating() (SelfMating method), 288
 SequentialParentChooser (built-in class), 282
 SequentialParentChooser() (SequentialParentChooser method), 282
 SequentialParentsChooser (built-in class), 282
 SequentialParentsChooser() (SequentialParentsChooser method), 282
 set() (RNG method), 290
 setAffected() (Individual method), 264
 setAllele() (Individual method), 264
 setAlleleLineage() (Individual method), 264
 setAncestralDepth() (Population method), 270
 setGenotype() (Individual method), 264
 setGenotype() (Population method), 270
 setIndInfo() (Population method), 270
 setInfo() (Individual method), 265
 setInfoFields() (Population method), 270
 setLineage() (Individual method), 265
 setLineage() (Population method), 270
 setOptions() (built-in function), 293
 setOptions() (in module simuOpt), 338
 setRNG, 21
 setRNG() (built-in function), 292
 setSex() (Individual method), 265
 setSubPopByIndInfo() (Population method), 270
 setSubPopName() (Population method), 270
 SettlementOfNewWorldModel (class in simuPOP.demography), 353
 SettlementOfNewWorldModel() (simuPOP.demography.SettlementOfNewWorldModel method), 353
 setVirtualSplitter() (Population method), 270
 sex() (Individual method), 265
 SexSplitter (built-in class), 277
 SexSplitter() (SexSplitter method), 277
 simuBackward() (simuPOP.utils.TrajectorySimulator method), 340
 simuForward() (simuPOP.utils.TrajectorySimulator method), 340
 simulateBackwardTrajectory() (in module simuPOP.utils), 341
 simulateForwardTrajectory() (in module simuPOP.utils), 341
 Simulator, 261
 Simulator (built-in class), 275
 Simulator() (Simulator method), 275
 simuOpt (module), 337
 simuPOP.demography (module), 347
 simuPOP.gsl (module), 360
 simuPOP.sampling (module), 354
 simuPOP.utils (module), 338
 snpMutate() (built-in function), 336
 SNPMutator (built-in class), 311
 SNPMutator() (SNPMutator method), 311
 sortIndividuals() (Population method), 271
 SplitEvent (class in simuPOP.demography), 350
 SplitEvent() (simuPOP.demography.SplitEvent method), 350
 splitSubPop() (Population method), 271
 SplitSubPops, 74
 SplitSubPops (built-in class), 302
 splitSubPops() (built-in function), 336
 SplitSubPops() (SplitSubPops method), 302
 Stat (built-in class), 321
 stat() (built-in function), 336
 Stat() (Stat method), 321
 stepwiseMutate() (built-in function), 336
 StepwiseMutator (built-in class), 309
 StepwiseMutator() (StepwiseMutator method), 309
 subPopBegin() (Population method), 271
 subPopByName() (Population method), 271
 subPopEnd() (Population method), 271
 subPopIndPair() (Population method), 271
 subPopName() (Population method), 271
 subPopNames() (Population method), 271
 subPopSize() (Population method), 272
 subPopSizes() (Population method), 271
 SummaryTagger (built-in class), 319
 SummaryTagger() (SummaryTagger method), 319
 swap() (Population method), 271
- ## T
- tagID() (built-in function), 336
 TerminateIf (built-in class), 330
 TerminateIf() (TerminateIf method), 330
 ThreeGenFamilySampler (class in simuPOP.sampling), 358
 ThreeGenFamilySampler() (simuPOP.sampling.ThreeGenFamilySampler method), 359
 TicToc (built-in class), 332
 TicToc() (TicToc method), 332
 totNumLoci() (GenoStruTrait method), 263
 traceRelatives() (Pedigree method), 274
 Trajectory (class in simuPOP.utils), 338
 Trajectory() (simuPOP.utils.Trajectory method), 339
 TrajectorySimulator (class in simuPOP.utils), 339

TrajectorySimulator() (simuPOP.utils.TrajectorySimulator method), [339](#)

transmitGenotype() (MendelianGenoTransmitter method), [304](#)

transmitGenotype() (Recombinator method), [307](#)

turnOffDebug() (built-in function), [293](#)

turnOnDebug() (built-in function), [293](#)

U

update() (simuPOP.utils.ProgressBar method), [342](#)

updateInfoFieldsFrom() (Population method), [271](#)

useAncestralGen() (Population method), [271](#)

V

vars() (Population method), [271](#)

vars() (Simulator method), [276](#)

viewVars() (in module simuPOP.utils), [342](#)

virtualSplitter() (Population method), [272](#)

vspByName() (BaseVspSplitter method), [276](#)

W

WeightedSampler (built-in class), [290](#)

WeightedSampler() (WeightedSampler method), [290](#)

WithArgs (built-in class), [289](#)

WithArgs() (WithArgs method), [289](#)

WithMode (built-in class), [289](#)

WithMode() (WithMode method), [289](#)