simuPOP User's Guide

Release 0.9.1 (Rev: 2459)

Bo Peng

December 2004

Last modified
February 19, 2009

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

Abstract

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

simuPOP is provided as a number of Python modules, which provide 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, simuPOP provides an increasing number of bundled scripts that perform simulations ranging from implementation of basic population genetics models to generating datasets under complex evolutionary scenarios. No knowledge about Python or simuPOP would be needed to run these simulations, if they happen to fit your need.

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 shows you how to use them in a simuPOP script. 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 Kimmal (2005) simuPOP: a forward-time population genetics simulation environment. *bioinformatics*, **21** (18): 3686-3687

Bo Peng and Christopher Amos (2008) Forward-time simulations of nonrandom mating populations using simuPOP. *bioinformatics*, **24** (11) 1408-1409.

Contents

List of Examples

Chapter 1

Introduction

1.1 What is simuPOP?

simuPOP is an 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. simuPOP provides a large number of Python objects and functions, and a mechanism to evolve populations forward in time. It is the users' responsibility to write a Python script to form a simulation. At a more user-friendly level, simuPOP provides an increasing number of built-in scripts so that users who are unfamiliar with Python and simuPOP can perform some pre-specified simulation processes. These scripts range from implementation of basic population genetics models to generating datasets under complex evolutionary scenarios. In addition, simuPOP modules and functions are provided to load and manipulate HapMap samples and to perform a number of popular gene-mapping methods.

Unlike other population genetics simulation applications that aim at specific evolutionary scenarios, simuPOP aims at providing a general purpose simulation program that can be used to write and study arbitrarily complex evolutionary scenarios. This makes simuPOP an ideal tool in a wide variety of applications ranging from demonstrating simple population genetics models to studying the evolution of complex human genetic diseases.

1.2 An overview of simuPOP concepts

A simuPOP **population** consists of individuals of the same **genotype structure**, which include properties such as number of homologous sets of chromosomes (ploidy), number of chromosomes, and names and locations of markers on each chromosome. Individuals can be divided into **subpopulations** that can be further divided into **virtual subpopulations** according to individual properties such as sex, affection status, or arbitrary auxiliary information such as age.

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 1.1), or to one or two parents 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 mating schemes, such as random, consanguineous, 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**.

simuPOP evolves a population generation by generation, following the evolutionary cycle depicted in Figure 1.1. Briefly speaking, a number of **pre-mating operators** such as a `mutator` are applied to a population before a mating scheme repeatedly chooses a parent or parents to produce offspring. **During-mating operators** such as *recombinator*

Figure 1.1: A life cycle of an evolutionary process

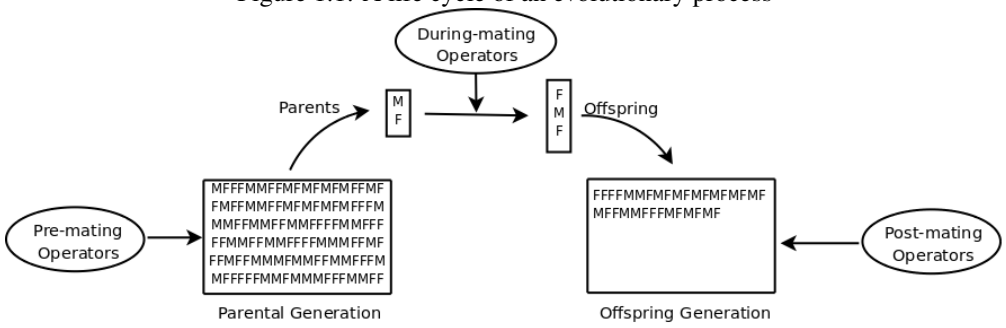


Illustration of the discrete-generation evolutionary model used by simuPOP.

can be used to adjust how offspring genotypes are formed from parental genotypes. After an offspring population is populated, **post-mating operators** can be applied, for example, to calculate population statistics. The offspring population will then become the parental population of the next evolutionary cycle.

Example 1.1: A simple example

```
>>> from simuPOP import *
>>> pop = population(size=1000, loci=[2])
>>> simu = simulator(pop, randomMating(), rep=3)
>>> simu.evolve(
...     preOps = [initByValue([1, 2, 2, 1])],
...     ops = [
...         recombinator(rate=0.01),
...         stat(LD=[0, 1]),
...         pyEval(r"'%.2f\t' % LD[0][1]", step=10),
...         pyOutput('\n', rep=-1, step=10)
...     ],
...     gen=100
... )
0.24    0.25    0.24
0.21    0.23    0.22
0.17    0.21    0.20
0.13    0.17    0.18
0.10    0.15    0.18
0.11    0.14    0.16
0.12    0.10    0.16
0.11    0.11    0.15
0.09    0.10    0.14
0.07    0.10    0.11
(100, 100, 100)
>>>
```

These concepts are demonstrated in Example 1.1, 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 third line creates a simulator with three replicates of this population. Random mating will be used to generate offspring. The last statement uses the `evolve()` function to evolve the populations for 100 generations, subject to five operators.

The first operator `initByValue` is applied to all populations before evolution. This operator initializes all individuals with the same genotype `12/21`. The other operators can be applied at every generation. `recombinator` is a during-mating operator that recombines parental chromosomes with the given recombination rate `0.01` during the generation of offspring; `stat` calculates linkage disequilibrium between the first and second loci. The results of this operator are stored in a local variable space of each population. The last two operators `pyEval` and `pyOutput` are

applied at the end of every 10 generations. `pyEval` is applied to all replicates to output calculated linkage disequilibrium values with a trailing tab, and the last operator outputs a newline after the last replicate. The result is a table of three columns, representing the decay of linkage disequilibrium of each replicate at 10 generation intervals. The return value of the `evolve` function, which is the number of evolved generations for each replicate, is also printed.

1.3 Features

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

1. simuPOP provides three types of modules that use 1, 8 or 16 bits to store an allele. The binary module (1 bit) is suitable for simulating a large number of SNP markers and the long module (16 bits) is suitable for simulating some population genetics models such as the infinite allele mutation model. simuPOP supports different types of chromosomes such as autosome, sex chromosomes and mitochondrial, with arbitrary number of markers.

2. 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 are used to track an individual's parents, and pack_year can be used to simulate an environmental factor associated with smoking.

3. simuPOP does not impose any limit on number of homologous sets of chromosomes, the size of the genome, or the number of individuals in a population. During an evolutionary process, a population can hold more than one most-recent generations. Pedigrees can be sampled from such multi-generation populations.

4. 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 user-provided function and mutates these alleles according to the returned values.

5. simuPOP provides more than 70 operators that cover all important aspects of genetic studies. These include mutation (*k*-allele, stepwise, generalized stepwise and hybrid), migration (arbitrary, can create new subpopulation), recombination and gene conversion (uniform or nonuniform, sex-specific), quantitative trait (single, multilocus or hybrid), selection (single-locus, additive, multiplicative or hybrid multi-locus models), penetrance (single, multi-locus or hybrid), ascertainment (case–control, affected sibpairs, random, nuclear and large pedigree), statistics calculation (including but not limited to allele, genotype, haplotype, heterozygote number and frequency; expected heterozygosity; bi-allelic and multi-allelic , and linkage disequilibrium measures), pedigree tracing, visualization (using R or other Python modules) and load/save in simuPOP's native format and many external formats such as Linkage.

6. Mating schemes and many operators 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. 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.

1.4 Installation

simuPOP is distributed under a GPL license and is hosted on `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), 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, MacOS X, and Sun Solaris 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. Besides a C++ compiler, several supporting tools and libraries are needed. Please refer to the `INSTALL` file for further information.

Thanks to the 'glue language' nature of Python, it is easy to inter-operate Python 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**. This technique is widely used in simuPOP so it is highly recommended that you install R and rpy is your are familiar with R. 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.

1.5 Getting help

1.5.1 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 1.2: Getting help using the `help()` function

```
>>> help(population.addInfoField)
Help on method population_addInfoField in module _simuPOP_std:

population_addInfoField(...) unbound simuPOP_std.population method
    Usage:

        x.addInfoField(field, init=0)

    Details:

        Add an information field field to a population and initialize its
        values to init.

>>>
```

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, at` etc are shared by all operators, and are explained in details only in class `baseOperator`.

1.5.2 Debug-related operators/functions

If your simuPOP session or script does not behave as expected, it might be helpful to let simuPOP print out some debug information. For example, the following code will 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 1.3: 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 right way to do this is to create a persistent population object:

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

You can use `TurnOnDebug(code)` and `TurnOffDebug(code)` to turn on and off debug information where `code` can be any debug code listed in `ListDebugCode()`. If you would like to turn on debugging during an evolutionary process, you can use operators `turnOnDebug` and `turnOffDebug`.

1.5.3 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 it. The PDF versions of both documents are distributed with simuPOP. You can also get the latest version of the documents online, from the simuPOP subversion repository

(http://simupop.sourceforge.net, click SF.net summary > Code > SVN Browse > trunk > doc). However, because simuPOP is under active development, there may be discrepancies between your local simuPOP installation and these latest documents.

A number of bundled scripts are distributed with simuPOP. They range from simple demonstration of population genetics models to observing the evolution of complex human genetic diseases. These scripts can be a good source to learn how to write a simuPOP script. Of course, if any of these scripts happens to fit your need, you may be able to use them directly, with writing a line of code.

A *simuPOP cookbook* is under development. The goal of this book is to provide recipes of commonly used simulation scenarios. A number of recipes are currently available under the doc/cookbook directory of a simuPOP distribution. This book might be made available online so that users can submit their own recipes.

If you cannot find the answer you need, or if you believe that you have located a bug, or if you would like to request a feature, please subscribe to the simuPOP mailinglist and send your questions there.

1.6 How to read this user's guide

This user's guide describes all simuPOP features using a lot of examples. Chapter 2 describes all classes in the simuPOP core. Although most topics and examples are simple, some topics need in-depth understanding of both the Python language and simuPOP. New simuPOP users can safely skip these sections. Chapter 3 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 4 describes features that are provided by various simuPOP utility modules. These modules provide extensions to the simuPOP core that greatly improves the usability and userfriendliness of simuPOP. The next chapter (Chapter 5) demonstrates how to write a script to solve a real-world simulation problem. The last chapter gives a quick description on some bundled scripts.

simuPOP is a comprehensive forward-time population genetics simulation environment with many unique features. If you are new to simuPOP, you can go through Chapter 2 and 3 quickly and understand what simuPOP is and what features it provide. Then, you can read Chapter 5 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 grow.

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 classes (e.g. population()), member functions (e.g. population.vars()) and parameter names start with small character and use capital character for the first character of each word afterward (e.g. population.subPopSize(), individual.setInfo()).
- Standalone functions start with capital character. This is how you can differ an operator from its function version. For example, TurnOnDebug(DBG_POPULATION) is the function to turn on debug mode for population related functions and turnOnDebug(DBG_POPULATION) will do nothing apparently, because it creates an operator.
- Constants start with Capital characters. 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:

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

  It usually possible to guess whether or not a parameter accepts a single or a list of objects by its name. For example, subPop accepts single subpopulation and subPops accepts a list of subpopulations.

Note: Plural form parameters usually accept single inputs. For example, `loci=1` can be used as a shortcut for `loci=[1]`. An exception to this convention is list of strings (e.g. parameter `infoFields=['smoke']`) because a single string such as `infoFields='smoke'` will be interpreted as `infoFields=['s', 'm', 'o', 'k', 'e']`.

Chapter 2

Core simuPOP components

2.1 Loading a simuPOP module

simuPOP consists of a number of Python modules, documents, tests and examples. Using Linux as an example, simuPOP installs the following files to your operating system:

- Core simuPOP modules (`simuPOP_XXX.py`, `_simuPOP_XXX.so`) and a number of utility modules (`simuUtil.py`, `simuOpt.py` etc) under `/usr/lib/python2X/site-packages`.
- `/usr/share/simuPOP/doc`: This directory contains the pdf version of this user's guide and the *simuPOP reference manual*.
- `/usr/share/simuPOP/test`: This directory contains all unit test cases. It is recommended that you test your simuPOP installation using these scripts if you compile simuPOP from source.
- `/usr/share/simuPOP/scripts`: This directory contains all the bundled scripts. It is worth noting that although these scripts are distributed with simuPOP, they are not tested as rigorously and as frequently as the simuPOP core. Please send an email to the simuPOP mailinglist if you notice any problem with them.

2.1.1 Short, long and binary modules and their optimized versions

There are six flavors of the core simuPOP module: short, long and binary 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 but not so if you need to simulate models such as the infinite allele model. In those cases, you should use the long allele version of the modules, which use 16 bits for each allele and can have 2^{16} possible allele states. On the other hand, 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. Despite of differences in internal memory layout, all these modules have the same interface.

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 all 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.

Example 2.1 and 2.2 demonstrate the differences between standard and optimized modules, by executing two invalid commands. A standard module returns proper error messages, while an optimized module returns erroneous results and or simply crashes.

Example 2.1: Use of standard simuPOP modules

```
>>> from simuPOP import *
simuPOP : Copyright (c) 2004-2008 Bo Peng
Version 0.9.2svn (Revision 2468, Feb 19 2009) for Python 2.4.3
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)]
Random Number Generator is set to mt19937 with random seed 0x34593880fd47c76f
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).
>>> pop = population(10, loci=[2])
>>> pop.locusPos(10)
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    # This script will genreate code/result pieces
IndexError: src/genoStru.h:558 absolute locus index (10) out of range of 0 - 1
>>> pop.individual(20).setAllele(1, 0)
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    # This script will genreate code/result pieces
IndexError: src/population.h:455 individual index (20) out of range of 0 ~ 9
>>>
```

Example 2.2: Use of optimized simuPOP modules

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

Example 2.2 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 `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 initial output. 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.

2.1.2 Random number generator

When simuPOP is loaded, it creates a default random number generator (`RNG`) of type `mt19937` using a random seed 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. `AvailableRNGs()`, `SetRNG(name, seed)`). It is also possible to save the random seed of a simuPOP session (c.f. `rng().seed()`) and use it to replay the session later. **Note:** `rng().seed()` returns the seed of the simuPOP random number generator. It can be used to replay your simulation if `rng()` 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(rng().seed())`.

2.1.3 Graphical user interface

There is no graphical user interface to simuPOP but various dialogs can be used for simple tasks. 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.simuOpt` is used to handle parameters. Other examples include class `simuUtil.simuProgress` that makes use of a progress dialog, and function `simuUtil.ViewVars` that uses a dialog to display a large number of variables. Note that the **use of GUI in simuPOP is optional in the sense that all functionalities can be achieved without a GUI**. For examples, `simuOpt.getParam()` will use a terminal to accept user input and `simuUtil.simuProgress` turns to a text-based progress bar in the non-GUI mode.

The use of GUI can be controlled either globally or individually. More specifically,

- By default, 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.
- If environmental variable `SIMUGUI` is set to `False`, no GUI will be used. If it is set to `Tkinter`, `Tkinter`-based dialogs will be used even if `wxPython` is available.
- The same parameters `True/False/wxPython/Tkinter` at the script level using command line option `--gui`. Note that `--gui=False` is commonly used to run scripts in batch mode.
- For each involved function or class, parameter `gui` is usually provided. The same set of options apply.

2.2 Pythonic issues

2.2.1 References and the `clone()` member function

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

```
pop = population()
pop1 = pop
```

will create 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 2.3. 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 2.3: 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 2.3 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 because we do not need to copy the first population from `simu` if it will be destroyed soon.

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)
```

2.2.2 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 the *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 2.4: Conversion between absolute and relative indexes

```
>>> pop = 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)
>>>
```

2.2.3 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 that can be used to iterate through all individuals, or all individuals in a (virtual) subpoulation. `simulator::populations()` can be used to iterate through all populations in a simulator. Example 2.14 demonstrates the use of ranges and iterators in simuPOP.

Example 2.5: Ranges and iterators

```
>>> pop = population(size=2, loci=[5, 6])
>>> InitByFreq(pop, [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),
...     print
...
0 2 0 1 0 2
2 0 2 2 2 2
>>>
```

2.2.4 `carray` datatype

simuPOP uses mostly standard Python types such as tuples, lists and dictionaries. However, for efficiency considerations, simuPOP defines and uses a new `carray` datatype to refer to an internal array of genotypes. Such an object can only be returned from `individual::genotype` and `population::genotype` functions. Instead of copying all genotypes to a Python tuple or list, these functions return a `carray` object that directly reflect the underlying genotype. This object behaves like a regular Python list except that the underlying genotype will be changed if elements of this object are changed. In addition, elements in this array will be changed if the underlying genotype is changed using another method.

Example 2.14 demonstrates the use of this datatype. It also shows how to get an independent list of alleles using the `list()` built-in function. Compare to `allele()`, `setAllele()` and `setGenotype()` functions, it is usually more efficient and more convenient to read and write genotypes using carray objects, although this usage is usually less readable.

Example 2.6: The carray datatype

```
>>> pop = population(size=2, loci=[3, 4])
>>> InitByFreq(pop, [0.3, 0.5, 0.2])
>>> ind = pop.individual(0)
>>> arr = ind.genotype()     # a carray to the underlying genotype
>>> geno = list(arr)         # a list of alleles
>>> print arr
[1, 1, 1, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1, 0]
>>> print geno
[1, 1, 1, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1, 0]
>>> arr.count(1)            # count
10
>>> arr.index(2)            # index
4
>>> ind.setAllele(5, 3)     # change underlying genotype using setAllele
>>> print arr               # arr is change
[1, 1, 1, 5, 2, 0, 1, 1, 1, 1, 1, 1, 1, 0]
>>> print geno              # but not geno
[1, 1, 1, 0, 2, 0, 1, 1, 1, 1, 1, 1, 1, 0]
>>> arr[2:5] = 4            # can use regular Python slice operation
>>> print ind.genotype()
[1, 1, 4, 4, 4, 0, 1, 1, 1, 1, 1, 1, 1, 0]
>>>
```

2.3 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()`).

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 2.14 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 2.7: Genotypic structure functions

```
>>> pop = population(size=[2, 3], ploidy=2, loci=[5, 10],
...     lociPos=[range(0, 5), range(0, 20, 2)], chromNames=['Chr1', 'Chr2'],
...     alleleNames=['A', 'C', 'T', 'G'])
>>> # access genotypic information from the 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)
'loc1-2'
>>> # utility functions
>>> ind.chromBegin(1)
5
>>> ind.chromByName('Chr2')
1
>>> # loci pos can be unordered within each chromosome
>>> pop = population(loci=[2, 3], lociPos=[3, 1, 1, 3, 2],
...     lociNames=['loc%d' % x for x in range(5)])
>>> pop.lociPos()
(1.0, 3.0, 1.0, 2.0, 3.0)
>>> pop.lociNames()
('loc1', 'loc0', 'loc2', 'loc4', 'loc3')
>>>
```

2.3.1 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 population would be enhanced over time.

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 2.14.

Example 2.8: An example of haplodiploid population

```
>>> pop = population(size=[2,5], ploidy=Haplodiploid, loci=[3, 5])
>>> InitByFreq(pop, [0.3, 0.7])
>>> Dump(pop)
Ploidy: 2 (haplodiploid)
Chromosomes:
1: chrom1 (Autosome, 3 loci)
```

```
  loc1-1 (1), loc1-2 (2), loc1-3 (3)
2: chrom2 (Autosome, 5 loci)
  loc2-1 (1), loc2-2 (2), loc2-3 (3), loc2-4 (4), loc2-5 (5)
population size: 7 (2 subpopulations with 2, 5 individuals)
Number of ancestral populations: 0

Subpopulation 0 (unnamed), 2 individuals:
   0: FU 111 11110 | 100 10101
   1: MU 111 10111 | ___ _____
Subpopulation 1 (unnamed), 5 individuals:
   2: FU 110 10110 | 111 00001
   3: FU 110 11110 | 111 01001
   4: FU 000 11101 | 111 11011
   5: FU 101 11110 | 111 11000
   6: FU 111 10110 | 110 10111
>>>
```

2.3.2 Autosomes, sex chromosomes, and other types of chromosomes

The default chromosome type is autosome, which is the *normal* chromosomes in diploid, and in haploid populations. simuPOP supports three other types of chromosomes, namely *ChromosomeX*, *ChromosomeY* and *Customized*. Sex chromosomes are only valid in haploid populations where chromosomes X and Y are used to determine the sex of an offspring. Customized chromosomes rely on user defined functions and operators to be passed from parents to offspring.

Example 2.14 shows how to specify different chromosome types, and how genotypes of these special chromosomes are arranged.

Example 2.9: Different chromosome types

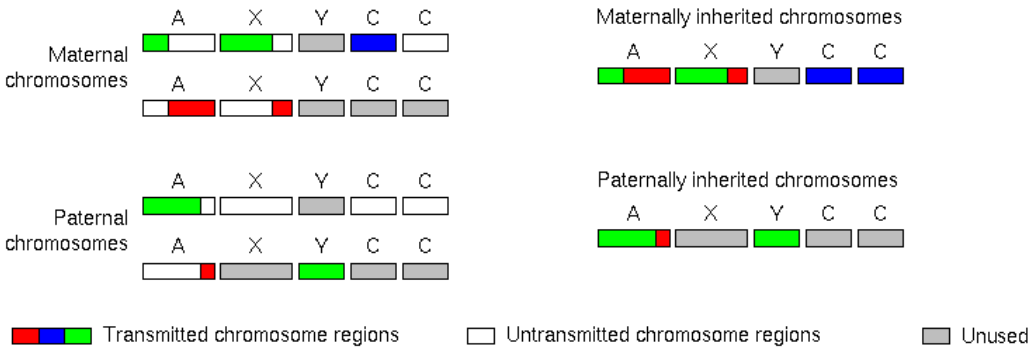
```
>>> pop = population(size=6, ploidy=2, loci=[3, 3, 6, 4, 4, 4],
...     chromTypes=[Autosome]*2 + [ChromosomeX, ChromosomeY] + [Customized]*2)
>>> InitByFreq(pop, [0.3, 0.7])
>>> Dump(pop, structure=False) # does not display genotypic structure information
Subpopulation 0 (unnamed), 6 individuals:
   0: FU 000 111 111101 ____ 1001 1011 | 000 010 100110 ____ 1111 1110
   1: MU 011 111 010111 ____ 0010 1010 | 111 110 ______ 1111 1111 0001
   2: FU 111 110 011111 ____ 0100 0111 | 110 111 111111 ____ 1111 1111
   3: FU 111 111 111110 ____ 0001 1111 | 011 111 011101 ____ 0000 1110
   4: MU 111 111 111011 ____ 1101 0111 | 110 110 ______ 0011 0111 0011
   5: FU 001 111 011110 ____ 1100 1011 | 111 111 111110 ____ 1100 1011
>>>
```

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.

As an advanced feature of simuPOP, chromosomes that do not follow the inheritance patterns of autosomes or sex chromosomes can be handled separately (see section 2.8.4 for an Example). Figure 2.1 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 2.1: 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.

2.3.3 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 relevant function. They are essential for the function 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 2.10: Basic usage of information fields

```
>>> pop = population(10, loci=[20], ancGen=1,
...     infoFields=['father_idx', 'mother_idx'])
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = [initByValue([0]*20+[1]*20)],
...     ops = [
...         parentsTagger(),
```

```
...           recombinator(rate=0.01)
...       ],
...       gen = 1
... )
(1,)
>>> pop = simu.extract(0)
>>> pop.indInfo('mother_idx')  # mother of all offspring
(7.0, 7.0, 4.0, 4.0, 8.0, 0.0, 2.0, 7.0, 6.0, 6.0)
>>> ind = pop.individual(0)
>>> mom = pop.ancestor(ind.intInfo('mother_idx'), 1)
>>> print ind.genotype(0)
[0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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]
>>> print mom.genotype(1)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

Example 2.14 demonstrates the basic usage of information fields. In this example, a population with two information fields `mother_idx` and `father_idx` are created. It can hold one ancestral generations (`ancGen=1`, see Section 2.5.5 for details) so the most recent parental generations will be kept in a population object. 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 index stored in this individual's information field. Genotypes at the first homologous copy of this individual's chromosome is printed, along with two parental chromosomes.

2.4 Individual

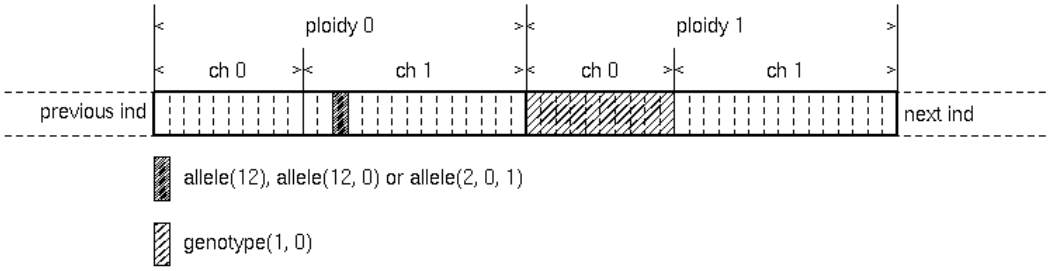
Individuals are building blocks of populations. An individual object cannot be created independently, but references to inidividuals can be retrieved using member functions of a population object. In addition to structural information shared by all individuals in a population, the individual class provides member functions to get and set *genotype*, *sex*, *affection status* and *information fields* of an individual. Example 2.12 demonstrates how to access and modify individual sex, affection status and information fields.

Example 2.11: Access Individual properties

```
>>> pop = 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 status,
>>> ind.sex()               # sex,
1
>>> ind.setInfo(4, 'x')     # and information fields
>>> ind.info('x')
4.0
>>>
```

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 2.2.

Figure 2.2: Memory layout of individual genotype



Single-allele read: allele(idx), allele(idx, p), allele(idx, p, ch)

Single-allele write: setAllele(allele, idx), setAllele(allele, idx, p), setAllele(allele, idx, p, ch)

Batch read: genotype(), genotype(p), genotype(p, ch)

Batch write: setGenotype(), setGenotype(p), setGenotype(p, ch)

simuPOP provides several functions to read/write individual genotype. 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. Modifying elements of this array will change the genotype of an individual directly. Only `count` and `index` list 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 2.12 demonstrates the use of these functions.

Example 2.12: Access individual genotype

```
>>> pop = 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]
>>>
```

2.5 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.

2.5.1 Subpopulations

A simuPOP population consists of one or more subpopulations. 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). You will rarely get a chance to use them directly because such operations are usually handled by operators.

Example 2.14 demonstrates how to use subpopulation related functions. Of particular interest is the `setSubPopByIndInfo()` function. This function takes an information field as parameter and rearrange individuals according to their values at this information field. Individuals with invalid (negative) values at this information field are removed. This is essentially how migration is implemented in simuPOP.

Example 2.13: Manipulation of subpopulations

```
>>> pop = 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])
>>> # split subpopulations
>>> pop.splitSubPops(1, [2, 7])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    # This script will genreate code/result pieces
AttributeError: 'population' object has no attribute 'splitSubPops'
>>> pop.subPopSizes()
(3, 9)
>>> # set information field to each individual's new subpopulation ID
>>> pop.setIndInfo([0, 1, 2, -1, 0, 1, 2, -1, -1, 0, 1, 2], 'x')
>>> # this manually triggers an migration, individuals with negative values
>>> # at this information field are removed.
>>> pop.setSubPopByIndInfo('x')
>>> Dump(pop, width=2, structure=False)
Subpopulation 0 (unnamed), 3 individuals:
   0: MU  0 |  0
```

```
   1: MU  4 |  0
   2: MU  9 |  0
Subpopulation 1 (unnamed), 3 individuals:
   3: MU  1 |  1
   4: MU  5 |  1
   5: MU 10 |  1
Subpopulation 2 (unnamed), 3 individuals:
   6: MU  2 |  2
   7: MU  6 |  2
   8: MU 11 |  2
>>>
```

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 2.14: Use of subpopulation names

```
>>> pop = population(size=[3, 4, 5], subPopNames=['x', 'y', 'z'])
>>> pop.removeSubPops([1])
>>> pop.subPopNames()
('x', 'z')
>>> pop.subPopByName('z')
1
>>> pop.splitSubPops(1, [2, 3])
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    # This script will genreate code/result pieces
AttributeError: 'population' object has no attribute 'splitSubPops'
>>> pop.subPopNames()
('x', 'z')
>>> pop.setSubPopName('z-1', 1)
>>> pop.subPopNames()
('x', 'z-1')
>>> pop.subPopByName('z')
Traceback (most recent call last):
  File "userGuide.py", line 1, in ?
    # This script will genreate code/result pieces
ValueError: build/population_std.cpp:213 Subpopulation z not found.
>>>
```

2.5.2 Virtual subpopulations

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. 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. 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 [spID, vspID] pair. Its name and size can be obtained using functions subPopName() and subPopSize(). Example 2.15 demonstrates how to apply virtual splitters to a population, and how to check VSP names and sizes.

Example 2.15: Define virtual subpopulations in a population

```
>>> import random
>>> pop = population(size=[200, 400], loci=[30], infoFields=['x'])
>>> # assign random information fields
>>> pop.setIndInfo([random.randint(0, 3) for x in range(pop.popSize())], 'x')
>>> # define a virtual splitter by information field 'x'
>>> pop.setVirtualSplitter(infoSplitter(field='x', values=[0, 1, 2, 3]))
>>> pop.numVirtualSubPop()     # Number of defined VSPs
4
>>> pop.subPopName([0, 0])     # Each VSP has a name
'unnamed - x = 0'
>>> pop.subPopSize([0, 0])     # Size of VSP 0 in subpopulation 0
50
>>> pop.subPopSize([1, 0])     # Size of VSP 0 in subpopulation 1
109
>>> # use a combined splitter that defines additional VSPs by sex
>>> InitSex(pop)
>>> pop.setSubPopName('subPop 1', 0)
>>> pop.setVirtualSplitter(combinedSplitter([
...     infoSplitter(field='x', values=[0, 1, 2, 3]),
...     sexSplitter()])
... )
>>> pop.numVirtualSubPop()     # Number of defined VSPs
6
>>> pop.subPopName([0, 4])     # VSP 4 is the first VSP defined by the sex splitter
'subPop 1 - Male'
>>> pop.subPopSize([0, 4])     # Number of male individuals
94
>>>
```

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 2.18 demonstrates how to initialize genotype and information fields to individuals in male and female VSPs.

Example 2.16: Applications of virtual subpopulations

```
>>> import random
>>> pop = population(10, loci=[2, 3], infoFields=['Sex'])
>>> InitSex(pop)
>>> pop.setVirtualSplitter(sexSplitter())
>>> # initialize male and females with different genotypes. Set initSex
>>> # to False because this operator will by default also initialize sex.
>>> InitByValue(pop, [[0]*5, [1]*5], subPops=([0, 0], [0, 1]), initSex=False)
>>> # set Sex information field to 0 for all males, and 1 for all females
>>> pop.setIndInfo([1], 'Sex', [0, 0])
>>> pop.setIndInfo([2], 'Sex', [0, 1])
>>> # Print individual genotypes, followed by values at information field Sex
>>> Dump(pop, structure=False)
Subpopulation 0 (unnamed), 10 individuals:
```

```
   0: FU 11 111 | 11 111 |  2
   1: FU 11 111 | 11 111 |  2
   2: FU 11 111 | 11 111 |  2
   3: FU 11 111 | 11 111 |  2
   4: FU 11 111 | 11 111 |  2
   5: MU 00 000 | 00 000 |  1
   6: MU 00 000 | 00 000 |  1
   7: FU 11 111 | 11 111 |  2
   8: MU 00 000 | 00 000 |  1
   9: MU 00 000 | 00 000 |  1
>>>
```

2.5.3 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 more then one generations are stored in a population, function `ancestor(idx, [subPop], gen)` can be used to access individual from an ancestral generation (see Section 2.5.5 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 2.18 demonstrates how to use all these individual-access functions.

Example 2.17: Access individuals of a population

```
>>> # create a population with two generations. The current generation has values
>>> # 0-9 at information field x, the parental generation has values 10-19.
>>> pop = population(size=[5, 5], loci=[2, 3], infoFields=['x'], ancGen=1)
>>> pop.setIndInfo(range(11, 20), 'x')
>>> pop1 = pop.clone()
>>> pop1.setIndInfo(range(10), 'x')
>>> pop.push(pop1)
>>> #
>>> ind = pop.individual(5)       # using absolute index
>>> ind.info('x')
5.0
>>> # use a for loop, and relative index
>>> for idx in range(pop.subPopSize(1)):
...     print pop.individual(idx, 1).info('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.info('x'),
...
5.0 6.0 7.0 8.0 9.0
>>> # Access individuals in VSPs
>>> pop.setVirtualSplitter(infoSplitter(cutoff=[3, 7], field='x'))
>>> for ind in pop.individuals([1, 1]):
...     print ind.info('x'),
...
```

```
5.0 6.0
>>> # Access individuals in ancetral generations
>>> pop.ancestor(5, 1).info('x')        # absolute index
16.0
>>> pop.ancestor(0, 1, 1).info('x')     # relative index
16.0
>>> # Or make ancestral generation the current generation and use 'individual'
>>> pop.useAncestralGen(1)
>>> pop.individual(5).info('x')         # absolute index
16.0
>>> pop.individual(0, 1).info('x')      # relative index
16.0
>>> # 'ancestor' can still access the 'present' (generation 0) generation
>>> pop.ancestor(5, 0).info('x')
5.0
>>>
```

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 2.18 demonstrates the use of such functions.

Example 2.18: Access individual properties in batch mode

```
>>> import random
>>> pop = population(size=[4, 6], loci=[2], infoFields=['x'])
>>> pop.setIndInfo([random.randint(0, 10) for x in range(10)], 'x')
>>> pop.indInfo('x')
(6.0, 5.0, 4.0, 1.0, 4.0, 1.0, 1.0, 6.0, 3.0, 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(infoSplitter(cutoff=[3], field='x'))
>>> pop.setGenotype([0])    # clear all values
>>> pop.setGenotype([5, 6, 7], [1, 1])
>>> pop.indInfo('x', 1)
(4.0, 1.0, 1.0, 6.0, 3.0, 10.0)
>>> pop.genotype(1)
[5, 6, 7, 5, 0, 0, 0, 0, 0, 0, 0, 0, 6, 7, 5, 6, 7, 5, 6, 7, 5, 6, 7, 5]
>>>
```

2.5.4 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`. Sections 2.3.3 and 2.5.3 have demonstrated how to read and write information fields from an individual, or from a population in batch mode.

Information fields can not located by their names or indexes. We have always used field names for clarity, at a cost of performance because these names have to be translated into indexes each time. When performance is a concern, you can use `idx=pop.infoIdx(name)` to get and use the index of an information field.

Example 2.19: Add and use of information fields in a population

```
>>> pop = population(10)
>>> pop.setInfoFields(['a', 'b'])
```

```
>>> pop.addInfoField('c')
>>> pop.addInfoFields(['d', 'e'])
>>> pop.infoFields()
('a', 'b', 'c', 'd', 'e')
>>> #
>>> cIdx = pop.infoIdx('c')
>>> eIdx = pop.infoIdx('e')
>>> pop.setIndInfo([1], cIdx)
>>> for ind in pop.individuals():
...     ind.setInfo(ind.info(cIdx) + 1, eIdx)
...
>>> print pop.indInfo(eIdx)
(2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0)
>>>
```

2.5.5 Ancestral populations

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 ascertainments.

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.

A typical use of ancestral generations is demonstrated in example 2.22. 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 2.20: Ancestral populations

```
>>> simu = simulator(population(500, loci=[1]), randomMating())
```

```
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         # start recording ancestral generations at generation 18
...         setAncestralDepth(2, at=[-2]),
...         stat(alleleFreq=[0], begin=-3),
...         pyEval(r"'%.3f\n' % alleleFreq[0][0]", begin=-3)
...     ],
...     gen = 20
... )
0.436
0.455
0.450
(20,)
>>> pop = simu.population(0)
>>> # start from current generation
>>> for i in range(pop.ancestralGens(), -1, -1):
...   pop.useAncestralGen(i)
...   Stat(pop, alleleFreq=[0])
...   print '%d   %.3f' % (i, pop.dvars().alleleFreq[0][0])
...
2   0.436
1   0.455
0   0.450
>>> # restore to the current generation
>>> pop.useAncestralGen(0)
>>>
```

2.5.6 Add and remove loci

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 2.22 demonstrates how to use these populations.

Example 2.21: Add and remove loci and chromosomes

```
>>> pop = 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')
>>> pop1 = 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], names=['rs7', 'rs8'])
(7, 10)
>>> # 1 1 1, 0 0 0, 2 0 2 0
>>> pop.removeLoci([8])
>>> Dump(pop)
Ploidy: 2 (diploid)
Chromosomes:
1: chr1 (Autosome, 3 loci)
  loc1-1 (1), loc1-2 (2), loc1-3 (3)
```

```
2: chr2 (Autosome, 3 loci)
  rs1 (0.5), rs2 (1), rs3 (2)
3: chr3 (Autosome, 4 loci)
  rs4 (1), rs7 (1.5), rs6 (3), rs8 (3.5)
population size: 10 (1 subpopulations with 10 individuals)
Number of ancestral populations: 0

Subpopulation 0 (unnamed), 10 individuals:
   0: MU 111 000 2020 | 111 000 2020
   1: MU 111 000 2020 | 111 000 2020
   2: MU 111 000 2020 | 111 000 2020
   3: MU 111 000 2020 | 111 000 2020
   4: MU 111 000 2020 | 111 000 2020
   5: MU 111 000 2020 | 111 000 2020
   6: MU 111 000 2020 | 111 000 2020
   7: MU 111 000 2020 | 111 000 2020
   8: MU 111 000 2020 | 111 000 2020
   9: MU 111 000 2020 | 111 000 2020
>>>
```

2.5.7 Population extraction

Another import population member function is `population::extract(field=None, loci=None, info=None, ancGen=-1, ped=None)`. It is a powerful function that can extract subset of individuals, loci, information fields and ancestral generations from an existing population. This function is widely used in ascertainment operators where individuals or pedigrees are extracted from an existing population and form a sample.

If all default parameters are used, this function is equivalent to `population::clone()`. If a list of loci or information fields are given to parameters `loci` and `info`, other specified loci and information fields will be copied to the extracted population. If a positive `ancGen` is given, only generations `0 - ancGen` will be extracted. The most interesting parameter is `ind`. Instead of given a list of individuals that will be extract, an information field is expected. This information field is expected to hold the new subpopulation ID to which each individual will belong in the extracted population. Individuals with negative values (invalid subpopulation ID) at this information field will not be extracted. If another population (or pedigree) with the same number of individuals is given, the information field from that population is used. Example 2.22 demonstrates the use of this powerful function.

Example 2.22: Extract individuals, loci and information fields from an existing population

```
>>> import random
>>> pop = population(size=[10, 10], loci=[5, 5],
...     infoFields=['x', 'y'])
>>> InitByValue(pop, range(10))
>>> pop.setIndInfo([-1]*4 + [0]*3 + [-1]*3 + [2]*4 + [-1]*3 + [1]*4, 'x')
>>> pop1 = pop.extract(field='x', loci=[1, 2, 3, 6, 7], infoFields=['x'])
>>> Dump(pop1, structure=False)
Subpopulation 0 (unnamed), 3 individuals:
   0: MU 123 67 | 123 67 |  0
   1: MU 123 67 | 123 67 |  0
   2: MU 123 67 | 123 67 |  0
Subpopulation 1 (unnamed), 3 individuals:
   3: MU 123 67 | 123 67 |  1
   4: MU 123 67 | 123 67 |  1
   5: FU 123 67 | 123 67 |  1
Subpopulation 2 (unnamed), 4 individuals:
   6: FU 123 67 | 123 67 |  2
   7: MU 123 67 | 123 67 |  2
   8: MU 123 67 | 123 67 |  2
   9: MU 123 67 | 123 67 |  2
```

```
>>>
```

2.5.8 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 `terminateIf` read 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 2.23 demonstrates how to set and access Population variables.

Example 2.23: Population variables

```
>>> from pprint import pprint
>>> pop = population(100, loci=[2])
>>> InitByFreq(pop, [0.3, 0.7])
>>> print pop.vars()    # No variable now
{}
>>> pop.dvars().myVar = 21
>>> print pop.vars()
{'myVar': 21}
>>> Stat(pop, popSize=1, alleleFreq=[0])
>>> # pprint prints in a less messy format
>>> pprint(pop.vars())
{'alleleFreq': [[0.32500000000000001, 0.67500000000000004]],
 'alleleNum': [[65, 135]],
 'myVar': 21,
 'numSubPop': 1,
 'popSize': 100,
 'subPop': [{'alleleFreq': [[0.32500000000000001, 0.67500000000000004]],
             'alleleNum': [[65, 135]],
             'popSize': 100}],
 'subPopSize': [100],
 'virtualPopSize': [100]}
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
135
>>> # use the dvars() function to access dictionary keys as attributes
>>> print pop.dvars().alleleNum[0][1]
135
>>> print pop.dvars().alleleFreq[0]
[0.32500000000000001, 0.67500000000000004]
>>>
```

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 `pyEval` operator in example 1.1 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 2.29 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.

Example 2.24: Expression evaluation in the local namespace of a population

```
>>> simu = simulator(population(100, loci=[1]),
...     randomMating(), 5)
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         stat(alleleFreq=[0]),
...         terminateIf('alleleFreq[0][0] == 0. or alleleFreq[0][0] == 1.')
...     ]
... )
(349, 364, 297, 85, 548)
>>>
```

2.5.9 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`. (Yes, it is `Load..` not `load..` because `LoadPopulation` is a global function.). **Virtual splitters are not saved** because they are considered as runtime definitions. Although files in any extension can be used, extension `.pop` is recommended.

The native simuPOP format is not human readable and is not recognized by other applications. Other formats such as the one used by the popular `FSTAT` software is supported. They are implemented in Python in a Python utility module `simuUtil.py`. simuPOP cannot use one of such formats because none of them can handle huge populations that simuPOP can handle, and unique features such as population variables. Example 2.29 demonstrates how to save and load a population in the native simuPOP format.

Example 2.25: Save and load a population

```
>>> pop = population(100, loci=[5], chromNames=['chrom1'])
>>> pop.dvars().name = 'my population'
>>> pop.save('sample.pop')
>>> pop1 = LoadPopulation('sample.pop')
>>> pop1.chromName(0)
'chrom1'
>>> pop1.dvars().name
'my population'
>>>
```

2.6 Operators

Operators are objects that act on populations. They can be used in the following ways:

- Operators are usually passed to the `ops`, `preOps` and `postOps` parameters the `evolve` function of a simulator. The simulator will apply these operators before (`preOps`), after (`postPos`) or during (`ops`) an evolutionary process. Depending on parameters of an operator, it can be applied before, during, and/or after mating in a life cycle of a generation (parameter `stage`, see Figure 1.1), to a subset of generations (parameters `begin`, `end`, `step`, `at`), a subset of populations in a simulator (parameter `rep`), a subset of (virtual) subpopulations in each replicate (parameter `subPop`).
- During-mating operators are used by mating schemes to transmit parental genotype (and sometimes information fields) to offspring. Applicability parameters such as `begin`, `end`, `rep` are ignored.

- Most of the operators can be applied to a population directly, using their function forms. Applicability parameters are ignored.

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

2.6.1 Applicable stages and generations

A simuPOP life cycle (a *generation*) can be divided into *pre-mating*, *during-mating* and *post-mating*. In the pre-mating stage, the present generation is the parental generation. In the during-mating stage, an offspring generation is populated from the parental generation. In the post-mating stage, the offspring generation has become the present generation. An operator can be applied at one or more stages at a life cycle. However, each operator has its own default value for the `stage` parameter and changes to this parameter are not always allowed. For example, a `recombinator` can only be applied `DuringMating` and it will ignore your attemp to apply it at another stage.

Operators that are passed to the `ops` parameter of the `simulator::evolve` function 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 the simulator has an ending generation, negative generations numbers 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 2.29 demonstrates how to set applicable stages and generations of an operator. In this example, a population is initialized before evolution using a `initByFreq` 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 2.26: Applicable stages and generations of an operator.

```
>>> simu = simulator(population(100, loci=[20]), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [
...         stat(alleleFreq=[0], begin=80, step=10),
...         pyEval(r"'After gen %d: allele freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             begin=80, step=10),
...         pyEval(r"'Around gen %d: allele Freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             at = [-10, -1], stage=PrePostMating)
...     ],
...     postOps = [savePopulation(output='sample.pop')],
...     gen=100
... )
After gen 80: allele freq: 0.28
Around gen 90: allele Freq: 0.28
After gen 90: allele freq: 0.19
Around gen 90: allele Freq: 0.19
Around gen 99: allele Freq: 0.19
Around gen 99: allele Freq: 0.19
(100,)
>>>
```

2.6.2 Applicable populations

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 `rep` 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 2.29 demonstrates how to use this `rep` 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 2.27: Apply operators to a subset of populations

```
>>> simu = simulator(population(100, loci=[20]), randomMating(), 5)
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [
...         stat(alleleFreq=[0], step=10),
...         pyEval('gen', step=10, rep=0),
...         pyEval(r"'\t%.2f' % alleleFreq[0][0]", step=10, rep=(0, 2, -1)),
...         pyOutput('\n', step=10, rep=-1)
...     ],
...     gen=30,
... )
0       0.27    0.20    0.14
10      0.33    0.29    0.09
20      0.28    0.38    0.07
(30, 30, 30, 30, 30)
>>>
```

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

2.6.3 Operator output

All operators we have seen, except for the `savePopulation` operator in Example 2.26, 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`, ...

Because a table output such as the one in Example 2.29 is written by several operators, it is clear that all of them need to use the `'>>'` output format.

The `savePopulation` operator in Example 2.26 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 2.29 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 R^2 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 2.28: Use the output and outputExpr parameters

```
>>> from simuPOP import *
>>> simu = simulator(population(size=1000, loci=[2]), randomMating(), rep=3)
>>> simu.evolve(
...     preOps = [initByValue([1, 2, 2, 1])],
...     ops = [
...         recombinator(rate=0.01),
...         stat(LD=[0, 1]),
...         pyEval(r"'%.2f\t' % LD[0][1]", step=20, output='>>LD.txt'),
...         pyOutput('\n', rep=-1, step=20, output='>>LD.txt'),
...         pyEval(r"'%.2f\t' % R2[0][1]", output='R2.txt'),
...         pyEval(r"'%.2f\t' % LD[0][1]", step=20, output="!'>>LD_%d.txt' % rep"),
...     ],
...     gen=100
... )
(100, 100, 100)
>>> print open('LD.txt').read()
0.25    0.24    0.24
0.20    0.21    0.19
0.15    0.16    0.16
0.15    0.15    0.12
0.11    0.12    0.10

>>> print open('R2.txt').read()     # Only the last write operation succeed.
0.07
>>> print open('LD_2.txt').read()   # Each replicate writes to a different file.
0.24    0.19    0.16    0.12    0.10
>>>
```

2.6.4 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.

For example, Example 2.29 defines a three-locus heterogeneity penetrance model [Risch, 1990] 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 2.29: Use a hybrid operator

```
>>> def myPenetrance(geno):
...     'A three-locus heterogeneity penetrance model'
...     if sum(geno) < 2:
...         return 0
...     else:
...         return sum(geno)*0.1
...
>>> simu = simulator(population(1000, loci=[20]*3), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.8, 0.2])],
...     ops = [
...         pyPenetrance(func=myPenetrance, loci=[10, 30, 50]),
...         stat(numOfAffected=True),
...         pyEval(r"'%d: %d\n' % (gen, numOfAffected)")
...     ],
...     gen = 5
... )
0: 78
1: 74
2: 76
3: 83
4: 79
(5,)
>>>
```

2.6.5 Python operators

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 `stage=DuringMating`), and can therefore perform arbitrary operations.

A pre- or post-mating `pyOperator` expects a function in the form of

```
func(pop [, param])
```

where `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`.

Example 2.30 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 `KamMutate` function is the function form of a mutator `kamMutator` (see Section 2.6.7 for details).

Example 2.30: A frequency dependent mutation operator

```
def dynaMutator(pop, param):
    '''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.'''
    # unpack parameter
    (cutoff, mu1, mu2) = param;
    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] < cutoff:
            KamMutate(pop, maxAllele=1, rate=mu1, loci=[i])
        else:
            KamMutate(pop, maxAllele=1, rate=mu2, loci=[i])
    return True
```

Example 2.50 demonstrates how to use this operator. It first initializes the population using two `initByFreq` 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 `stage=PreMating` (the default stage is post mating) so allele frequencies have to be recalculated to be used by post-mating operator `pyEval`.

Example 2.31: Use a pyOperator during evolution

```
>>> simu = simulator(population(size=10000, loci=[2, 3]),
...     randomMating())
>>> simu.evolve(
...     preOps = [
...         initByFreq([.99, .01], loci=[0, 2, 4]),
...         initByFreq([.8, .2], loci=[1, 3])],
...     ops = [
...         pyOperator(func=dynaMutator, param=(.2, 1e-2, 1e-5), stage=PreMating),
...         stat(alleleFreq=range(5), step=10),
...         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
0.10 0.20 0.11 0.20 0.11
0.17 0.21 0.18 0.21 0.17
0.20 0.21 0.21 0.20 0.21
(31,)
>>>
```

An `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 on parameter `param` and `offspringOnly`, such an operator accepts a function in the form of

```
func(pop, dad, mom, off [, param])  # if offspringOnly=False (default)
func(off [, param])                 # if offspringOnly=True
```

Example 2.32 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 2.32: Use a during-mating pyOperator

```
>>> def rejectInd(off):
```

```
...     'reject an individual if it off.allele(0) == 1'
...     return off.allele(0) == 0
...
>>> simu = simulator(population(size=100, loci=[1]),
...     randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         pyOperator(func=rejectInd, stage=DuringMating, offspringOnly=True),
...     ],
...     gen = 1
... )
(1,)
>>> # You should see no individual with allele 1 at locus 0, ploidy 0.
>>> simu.population(0).genotype()[:20]
[0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1]
>>>
```

`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 2.8.8 describes how to do this.

2.6.6 Define your own operators

This is an advanced topic of simuPOP. New simuPOP users can safely skip this section.

`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 `simuRPy.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 2.50 rewrites the `dynaMutator` defined in Example 2.30 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 2.33: Define a new Python operator

```
>>> class dynaMutator(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.'''
...     def __init__(self, cutoff, mu1, mu2, *args, **kwargs):
...         self.cutoff = cutoff
...         self.mu1 = mu1
...         self.mu2 = mu2
...         pyOperator.__init__(self, func=self.mutate, *args, **kwargs)
...     #
...     def mutate(self, pop):
...         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:
```

```
...                 KamMutate(pop, maxAllele=1, rate=self.mu1, loci=[i])
...             else:
...                 KamMutate(pop, maxAllele=1, rate=self.mu2, loci=[i])
...         return True
...
>>> simu = simulator(population(size=10000, loci=[2, 3]),
...     randomMating())
>>> simu.evolve(
...     preOps = [
...         initByFreq([.99, .01], loci=[0, 2, 4]),
...         initByFreq([.8, .2], loci=[1, 3])],
...     ops = [
...         dynaMutator(cutoff=.2, mu1=1e-2, mu2=1e-5, stage=PreMating),
...         stat(alleleFreq=range(5), step=10),
...         pyEval(r"' '.join(['%.2f' % alleleFreq[x][1] for x in range(5)]) + '\n'",
...             step=10),
...     ],
...     gen = 31
... )
0.02 0.21 0.02 0.20 0.02
0.11 0.21 0.11 0.21 0.11
0.19 0.20 0.16 0.21 0.18
0.21 0.22 0.21 0.22 0.20
(31,)
>>>
```

New during-mating operators can be defined similarly. They are usually used to define customized genotype transmitters. Section 2.8.5 will describe this feature in detail.

2.6.7 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 `initByFreq` operator and apply to a population as follows:

```
initByFreq([.3, .2, .5]).apply(pop)
```

Similarly, you can apply the hybrid penetrance model defined in Example 2.29 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 `InitByFreq` is defined for operator `initByFreq` as follows

Example 2.34: The function form of operator `initByFreq`

```
>>> def InitByFreq(pop, *args, **kwargs):
...     initByFreq(*args, **kwargs).apply(pop)
...
>>> InitByFreq(pop, [.2, .3, .5])
>>>
```

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

```
InitByFreq(pop, [.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.

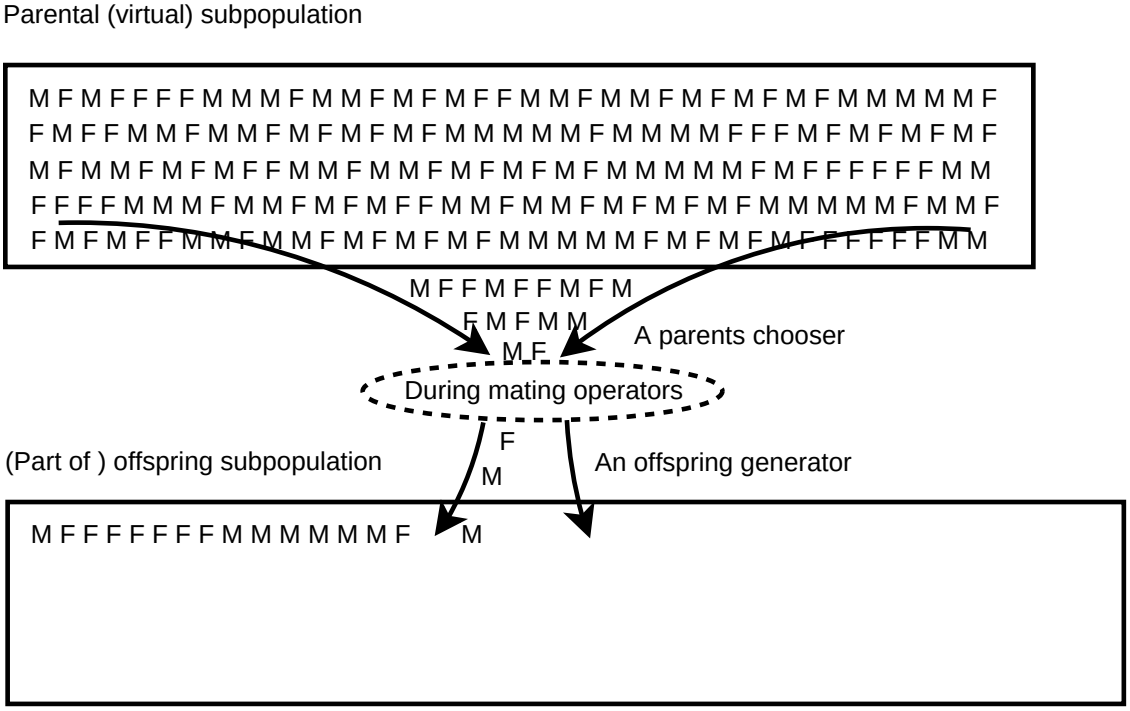
2.7 Mating Schemes

Mating schemes are responsible for populating an offspring generation from the parental generation. There are currently three 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 2.3 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** that follows a recorded evolutionary scenario. The selection of parents and the production of offspring are controlled by a pedigree. This mating scheme does not support virtual subpopulation.

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 2.3: 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.

2.7.1 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 follow-

ing rules:

- 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 a migrator, can change population or subpopulation sizes.
- If fixed subpopulation sizes are given to parameter `subPopSize`. A mating scheme will generation 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 two parameters: the generation number and the current subpopulation sizes, and return an array of new subpopulation sizes. A single number can be returned if there is only one subpopulation.

The following examples demonstrate these cases. Example 2.35 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 2.35: Free change of subpopulation sizes

```
>>> simu = simulator(
...     population(size=[500, 1000], infoFields=['migrate_to']),
...     randomMating())
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[797, 703]
[918, 582]
[980, 520]
(3,)
>>>
```

Example 2.36 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 2.36: Force constant subpopulation sizes

```
>>> simu = simulator(
...     population(size=[500, 1000], infoFields=['migrate_to']),
...     randomMating(subPopSize=[500, 1000]))
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...         stat(popSize=True, stage=PrePostMating),
...         pyEval(r'"%s\n" % subPopSize', stage=PrePostMating)
...     ],
...     gen = 3
... )
[818, 682]
[500, 1000]
```

```
[790, 710]
[500, 1000]
[770, 730]
[500, 1000]
(3,)
>>>
```

Example 2.41 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 2.37: Use a demographic function to control population size

```
>>> def demo(gen, oldSize=[]):
...     return [500 + gen*10, 1000 + gen*10]
...
>>> simu = simulator(
...     population(size=[500, 1000], infoFields=['migrate_to']),
...     randomMating(subPopSize=demo))
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         migrator(rate=[[0.8, 0.2], [0.4, 0.6]]),
...         stat(popSize=True),
...         pyEval(r'"%s\n" % subPopSize')
...     ],
...     gen = 3
... )
[500, 1000]
[510, 1010]
[520, 1020]
(3,)
>>>
```

All these examples have fixed number of subpopulations. Section **??** will introduce how to split and merge subpopulations dynamically.

2.7.2 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 will be passed to this function, which allows different numbers of offspring at different generations.
- If a tuple (or list) with more than one numbers is given, the first number must be one of `GeometricDistribution`, `PoissonDistribution`, `BinomialDistribution` and `UniformDistribution`, with one or two additional parameters. The number of offspring will then follow a specific statistical distribution. Note that all these distributions are adjusted so that the minimal number of offspring is 1.

More specifically,

- `numOffspring=(GeometricDistribution, p)`: The number of offspring for each mating event follows a geometric distribution with mean $1/p$ and variance $(1-p)/p^2$:

$$\Pr(k) = p(1-p)^{k-1} \quad \text{for } k \geq 1$$

- `numOffspring=(PoissonDistribution, p)`: The number of offspring for each mating event follows a shifted Poisson distribution with mean $p+1$ (you need to specify, for example, 2, if you want a mean family size of 3) and variance p. The distribution is

$$\Pr(k) = p^{k-1}\frac{e^{-p}}{(k-1)!} \quad \text{for} k \geq 1$$

- `numOffspring=(BinomialDistribution, p, n)`: The number of offspring for each mating event follows a shifted Binomial distribution with mean $(n-1)p+1$ and variance $(n-1)p(1-p)$.

$$\Pr(k) = \frac{(n-1)!}{(k-1)!(n-k)!}p^{k-1}(1-p)^{n-k}+1 \quad \text{for } n \geq k \geq 1$$

- `numOffspring=(UniformDistribution, a, b)`: The number of offspring for each mating event follows a discrete uniform distribution with lower bound a and upper bound b.

$$\Pr(k) = \frac{1}{b-a+1} \quad \text{for } b \geq k \geq a$$

Example 2.38 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 2.38: Control the number of offspring per mating event.

```
>>> def checkNumOffspring(ms):
...     '''Check the number of offspring for each family using
...        information field father_idx
...     '''
...     simu = simulator(
...         population(size=[30], infoFields=['father_idx', 'mother_idx']),
...         matingScheme=ms)
...     simu.evolve(
...         preOps = [initSex()],
...         ops=[parentsTagger()],
...         gen=1)
...     # get the parents of each offspring
...     parents = [(x, y) for x, y in zip(simu.population(0).indInfo('mother_idx'),
...         simu.population(0).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(randomMating(numOffspring=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(randomMating(numOffspring=func))
[6, 8, 5, 8, 3]
>>> # Case 3: A geometric distribution
>>> checkNumOffspring(randomMating(numOffspring=(GeometricDistribution, 0.3)))
[2, 1, 16, 4, 2, 3, 2]
>>> # Case 4: A Possition distribution
>>> checkNumOffspring(randomMating(numOffspring=(PoissonDistribution, 3)))
[4, 2, 4, 4, 5, 4, 1, 2, 4]
>>> # Case 5: A Binomial distribution
>>> checkNumOffspring(randomMating(numOffspring=(BinomialDistribution, 0.1, 10)))
[4, 1, 2, 2, 1, 3, 2, 3, 2, 1, 4, 3, 2]
>>> # Case 6: A uniform distribution
>>> checkNumOffspring(randomMating(numOffspring=(UniformDistribution, 2, 6)))
[2, 4, 4, 2, 5, 3, 2, 5, 3]
>>>
```

2.7.3 Determine offspring sex

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=NoSex`: Sex is not simulated so everyone is Male. This is the default mode where offspring can be Male or Female with equal probability.
- `sexMode=(ProbOfMale, prob)`: Produce males with given probability.
- `sexMode=(NumOfMale, n)`: The first `n` offspring in each family will be `Male`. If the number of offspring at a mating event is less than or equal to `n`, all offspring will be male.
- `sexMode=(NumOfFemale, n)`: The first `n` offspring in each family will be `Female`.

`NumOfMale` and `NumOfFemale` 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 2.39 demonstrates how to use parameter `sexMode`. In this example, a function `checkSexMode` is defined. It takes a mating scheme as its input parameter and use it to evolve a population with 40 individuals. After evolving a population for one generation, sexes of all offspring are returned as a string.

Example 2.39: Determine the sex of offspring

```
>>> def checkSexMode(ms):
...     '''Check the assignment of sex to offspring'''
...     simu = simulator(
...         population(size=[40]),
...         matingScheme=ms)
```

```
...     simu.evolve(preOps = [initSex()], ops=[], gen=1)
...     # return individual sex as a string
...     return ''.join([ind.sexChar() for ind in simu.population(0).individuals()])
...
>>> # Case 1: NoSex (all male, randomMating will not continue)
>>> checkSexMode(randomMating(sexMode=NoSex))
'MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM'
>>> # Case 2: RandomSex (Male/Female with probability 0.5)
>>> checkSexMode(randomMating(sexMode=RandomSex))
'FMFFFMFFFMFFFMFMMFFMFFFMMFMFMMFFMFFMFMMM'
>>> # Case 3: ProbOfMale (Specify probability of male)
>>> checkSexMode(randomMating(sexMode=(ProbOfMale, 0.8)))
'FMMMMMFFMMMMFMMMMMMMMFFMMMMMMMMFMMFMMMMM'
>>> # Case 4: NumOfMale (Specify number of male in each family)
>>> checkSexMode(randomMating(numOffspring=3, sexMode=(NumOfMale, 1)))
'MFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFM'
>>> # Case 5: NumOfFamel (Specify number of female in each family)
>>> checkSexMode(randomMating(
...     numOffspring=(UniformDistribution, 4, 6),
...     sexMode=(NumOfFemale, 2))
... )
'FFMMMFFMMMMFFMMMFFMMMFFMMMFFMMMFFMMMFFMMMFFM'
>>>
```

2.7.4 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 2.40 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=(NumOfMale, 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 2.40: Sexual monogamous mating

```
>>> simu = simulator(population(20, infoFields=['father_idx', 'mother_idx']),
...     monogamousMating(numOffspring=2, sexMode=(NumOfMale, 1)))
>>> simu.evolve(
...     preOps = [initSex(sex=(Male, Female))],
...     ops = [parentsTagger()],
...     gen = 5
... )
(5,)
>>> pop = simu.extract(0)
>>> [ind.sex() for ind in pop.individuals()]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> [ind.intInfo('father_idx') for ind in pop.individuals()]
[2, 2, 8, 8, 18, 18, 4, 4, 16, 16, 6, 6, 14, 14, 10, 10, 0, 0, 12, 12]
```

```
>>> [ind.intInfo('mother_idx') for ind in pop.individuals()]
[19, 19, 17, 17, 9, 9, 11, 11, 5, 5, 3, 3, 15, 15, 7, 7, 1, 1, 13, 13]
>>> # count the number of distinct parents
>>> len(set(pop.indInfo('father_idx')))
10
>>> len(set(pop.indInfo('mother_idx')))
10
>>>
```

2.7.5 Polygamous mating

In comparison to monogamous mating, parents in a polygamous mate with more than one spouse during their life-cycle. Both *polygany* (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 2.41 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 2.41: Sexual polygamous mating

```
>>> simu = simulator(population(100, infoFields=['father_idx', 'mother_idx']),
...     polygamousMating(polySex=Male, polyNum=2))
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [parentsTagger()],
...     gen = 5
... )
(5,)
>>> pop = simu.extract(0)
>>> [ind.intInfo('father_idx') for ind in pop.individuals()][:20]
[33, 33, 21, 21, 86, 86, 67, 67, 53, 53, 7, 7, 79, 79, 42, 42, 43, 43, 97, 97]
>>> [ind.intInfo('mother_idx') for ind in pop.individuals()][:20]
[32, 0, 65, 26, 52, 5, 44, 63, 0, 12, 52, 98, 45, 30, 14, 30, 75, 44, 65, 92]
>>>
```

2.7.6 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 2.42 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 2.42: Asexual random mating

```
>>> simu = simulator(population(100, ploidy=1, loci=[5, 5], ancGen=1,
...     infoFields=['parent_idx']),
...     randomSelection())
>>> simu.evolve(
...     preOps = [initByFreq([0.3, 0.7])],
...     ops = [parentTagger()],
```

```
...     gen = 5
... )
(5,)
>>> pop = simu.extract(0)
>>> ind = pop.individual(0)
>>> par = pop.ancestor(ind.intInfo('parent_idx'), 1)
>>> print ind.sex(), ind.genotype()
2 [1, 1, 1, 1, 0, 0, 0, 0, 0, 1]
>>> print par.sex(), par.genotype()
2 [1, 1, 1, 1, 0, 0, 0, 0, 0, 1]
>>>
```

2.7.7 Mating with alpha individuals

The `alphaMating` mating scheme is intended to simulate animal populations in which only individuals with alpha status have the power to mate. In this mating scheme, a number of alpha individuals with specified sex (`alphaSex`) are determined, either randomly (`alphaNum`) or according to values at an information field (`alphaField`). During mating, only individuals from this alpha group can be selected to mate.

Example 2.43 gives a simple evolutionary scenario where two alpha males are chosen according to individual fitness values at each generation. The fitness value of each individual is determined by his/her genotype at the first locus, `0.8`, `0.8`, and `1` for genotype `AA`, `Aa`, and `aa` respectively. Because individuals having mutant `a` have a high probability to be selected, and become the alpha male in this population, the frequency of this mutant tend to increase in this population.

Example 2.43: Random mating with alpha individuals

```
>>> simu = simulator(population(1000, loci=[5],
...     infoFields=['father_idx', 'mother_idx', 'fitness']),
...     alphaMating(alphaSex=Male, alphaNum=2))
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [parentsTagger(),
...         maSelector(loci=[0], fitness=[0.8, 0.8, 1]),
...         stat(alleleFreq=[0]),
...         pyEval(r'"%.2f\n" % alleleFreq[0][1]', step=5)
...     ],
...     gen = 20,
... )
0.64
0.73
0.68
0.82
(20,)
>>> pop = simu.extract(0)
>>> [ind.intInfo('father_idx') for ind in pop.individuals()][:10]
[688, 688, 498, 498, 688, 498, 688, 498, 498, 498]
>>> [ind.intInfo('mother_idx') for ind in pop.individuals()][:10]
[992, 643, 788, 500, 954, 534, 417, 663, 90, 200]
>>>
```

2.7.8 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 `haplodiplodMating`. 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 2.44 demonstrates how to use this mating scheme. It uses three initializers because sex has to be initialized before two other intializers can initialize genotype by sex.

Example 2.44: Random mating in haplodiploid populations

```
>>> pop = population(10, ploidy=Haplodiploid, loci=[5, 5],
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.setVirtualSplitter(sexSplitter())
>>> simu = simulator(pop, haplodiploidMating())
>>> simu.evolve(
...     preOps = [initSex(),
...         initByValue([0]*10, subPops=[(0, 0)], initSex=False),
...         initByValue([1]*10+[2]*10, subPops=[(0, 1)], initSex=False)],
...     ops = [parentsTagger(),
...         dumper(structure=False, stage=PrePostMating)],
...     gen = 1
... )
Subpopulation 0 (unnamed), 10 individuals:
   0: MU 00000 00000 | _____ _____ |  0 0
   1: FU 11111 11111 | 22222 22222 |  0 0
   2: MU 00000 00000 | _____ _____ |  0 0
   3: MU 00000 00000 | _____ _____ |  0 0
   4: FU 11111 11111 | 22222 22222 |  0 0
   5: MU 00000 00000 | _____ _____ |  0 0
   6: MU 00000 00000 | _____ _____ |  0 0
   7: MU 00000 00000 | _____ _____ |  0 0
   8: FU 11111 11111 | 22222 22222 |  0 0
   9: FU 11111 11111 | 22222 22222 |  0 0
Subpopulation 0 (unnamed), 10 individuals:
   0: FU 11111 11111 | 00000 00000 |  0 4
   1: FU 11111 11111 | 00000 00000 |  2 4
   2: MU 11111 22222 | _____ _____ |  0 8
   3: FU 22222 11111 | 00000 00000 |  2 8
   4: FU 22222 11111 | 00000 00000 |  7 8
   5: MU 22222 11111 | _____ _____ |  7 8
   6: FU 11111 11111 | 00000 00000 |  3 4
   7: FU 11111 11111 | 00000 00000 |  2 9
   8: MU 11111 22222 | _____ _____ |  5 8
   9: MU 22222 11111 | _____ _____ |  6 1
(1,)
>>>
```

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.

2.7.9 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 2.45 initializes each chromosome with different alleles to demonstrate how these alleles are transmitted in this population.

Example 2.45: Selfing mating scheme

```
>>> pop = population(20, loci=[8])
>>> # every chromosomes are different. :-)
```

```
>>> for idx, ind in enumerate(pop.individuals()):
...     ind.setGenotype([idx*2], 0)
...     ind.setGenotype([idx*2+1], 1)
...
>>> simu = simulator(pop, selfMating())
>>> simu.evolve(
...     ops = [recombinator(rate=0.1)],
...     gen = 1
... )
(1,)
>>> Dump(simu.population(0), width=3, structure=False, max=10)
Subpopulation 0 (unnamed), 20 individuals:
   0: FU   3  3  3  3  3  3  3  3 |   3  3  3  3  3  3  3  3
   1: FU  17 17 17 17 17 17 16 16 |  17 17 17 17 17 17 17 17
   2: FU  33 33 33 33 32 32 32 32 |  32 32 32 32 32 32 33 33
   3: FU   9  9  9  9  9  9  9  9 |   8  8  8  8  9  9  9  9
   4: FU  20 20 20 20 20 20 21 21 |  21 21 21 21 21 21 21 21
   5: MU  20 20 21 21 21 21 21 20 |  20 20 20 20 20 21 21 21
   6: MU   3  3  3  3  3  3  2  2 |   3  3  3  3  3  3  3  3
   7: MU  25 25 25 25 25 24 24 24 |  25 24 25 25 25 25 25 25
   8: MU  12 13 13 13 13 13 13 13 |  12 12 12 12 12 12 12 12
   9: MU  10 10 10 10 10 10 10 10 |  11 11 11 11 10 10 10 11
>>>
```

2.7.10 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.

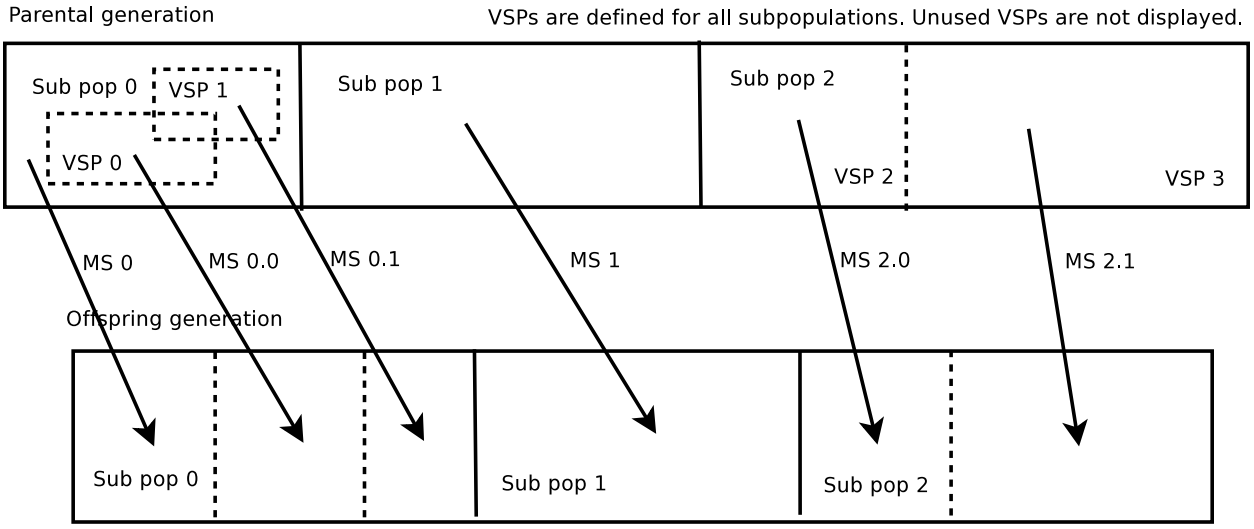
A heterogeneous mating scheme (`heteroMating`) accepts a list of mating schemes that are applied to different subpopulation or virtual subpopulations. If multiple mating schemes are applied to the same subpopulation, each of them only population part of the offspring subpopulation. This is illustrated in Figure 2.4.

For example, Example 2.46 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 2.46: Applying different mating schemes to different subpopulations

```
>>> pop = population(size=[1000, 1000], loci=[2],
...     infoFields=['father_idx', 'mother_idx'])
>>> simu = simulator(pop, heteroMating(
...     [randomMating(numOffspring=2, subPop=0),
...      randomMating(numOffspring=4, subPop=1)
...     ])
... )
>>> simu.evolve(
...     preOps = [initSex()],
...     ops= [parentsTagger()],
...     gen=10
... )
(10,)
>>> pop = simu.extract(0)
>>> [ind.intInfo('father_idx') for ind in pop.individuals(0)][:10]
[682, 682, 783, 783, 546, 546, 528, 528, 990, 990]
>>> [ind.intInfo('father_idx') for ind in pop.individuals(1)][:10]
[1191, 1191, 1191, 1191, 1913, 1913, 1913, 1913, 1064, 1064]
```

Figure 2.4: 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.

```
>>>
```

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 2.47 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 2.47: Applying different mating schemes to different virtual subpopulations

```
>>> pop = population(size=[1000], loci=[2],
...     infoFields=['father_idx', 'mother_idx'])
>>> pop.setVirtualSplitter(proportionSplitter([0.2, 0.8]))
>>> simu = simulator(pop, heteroMating(
...     matingSchemes = [
...         selfMating(subPop=(0, 0)),
...         randomMating(subPop=(0, 1))
...     ])
... )
>>> simu.evolve(
...     preOps = [initSex()],
```

```
...     ops= [parentsTagger()],
...     gen = 10
... )
(10,)
>>> pop = simu.extract(0)
>>> [ind.intInfo('father_idx') for ind in pop.individuals(0)][:15]
[49, 127, 141, 21, 66, 6, 78, 50, 26, 22, 111, 86, 86, 36, 38]
>>> [ind.intInfo('mother_idx') for ind in pop.individuals(0)][:15]
[132, 82, 143, -1, -1, -1, 105, 194, -1, -1, 173, 3, 110, 45, -1]
>>>
```

Because there is no restriction on the choice of VSPs, mating schemes can be applied to overlapped (virtual) subpopulations. For example,

```
heteroMating(
    matingSchemes = [
        selfMating(subPop=(0, 0)),
        randomMating(subPop=0)
        ]
)
```

will apply selfMating to the first 20% individuals, and randomMating will be applied to all individuals. Similarly,

```
heteroMating(
    matingSchemes = [
        selfMating(subPop=0),
        randomMating(subPop=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 position 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 N 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 $\frac{10}{23}N$, $\frac{5}{23}N$, $\frac{8}{23}N$ 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 $N \neq 2400$ 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 $\frac{1}{6}N$, $\frac{2}{6}N$, $\frac{1}{3}N$ 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, $\frac{2}{5}(N-500)$, $\frac{3}{5}(N-500)$ individuals respectively.

The last case is demonstrated in Example 2.48 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 2.48: A weighting scheme used by heterogeneous mating schemes.

```
>>> pop = population(size=[1000], loci=[2],
...     infoFields=['mark'])
>>> pop.setVirtualSplitter(rangeSplitter([[0, 500], [200, 1000]]))
>>> def markOff(param):
...     '''define a Python during mating operator that marks
...        individual information field 'mark'
...     '''
...     def func(off, param):
...         off.setInfo(param, 'mark')
...         return True
...     return pyOperator(func=func, param=param, stage=DuringMating,
...         offspringOnly=True)
...
>>> simu = simulator(pop, heteroMating(
...     matingSchemes = [
...         randomMating(subPop=0, weight=-0.5, ops=[markOff(0)]),
...         randomMating(subPop=(0, 0), weight=2, ops=[markOff(1)]),
...         randomMating(subPop=(0, 1), weight=3, ops=[markOff(2)])
...     ])
... )
>>> simu.evolve(
...     preOps = [initSex()],
...     ops= [],
...     gen = 10
... )
(10,)
>>> marks = list(simu.extract(0).indInfo('mark'))
>>> marks.count(0.)
500
>>> marks.count(1.)
200
>>> marks.count(2.)
300
>>>
```

2.8 Non-random and customized mating schemes

The following sections discuss advanced topics of simuPOP. New simuPOP users can safely skip these sections.

2.8.1 The structure of a homogeneous mating scheme

A *homogeneous mating scheme* 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**.
4. A offspring generator uses one or more during-mating operators to transmit parental genotype to offspring. These operators are call **genotype transmitters**.
5. After the offspring generation is populated, it will replace the parental generation and becomes the present generation of a population.

A simuPOP mating scheme uses a particular set of parent chooser, offspring generator, and genotype transmitters. 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.

Example 2.49 demonstrates how the most commonly used mating scheme, the diploid sexual `randomMating` mating scheme is defined in `simuPOP.py`. The following sections basically explain how you can construct your own mating scheme from scratch, using stocked or customized parent chooser, offspring generator and genotype transmitters.

Example 2.49: Define a random mating scheme

```
def mendelianOffspringGenerator(ops=[], *args, **kwargs):
    'An offspring generator that uses mendelianGenoTransmitter()'
    return  offspringGenerator([mendelianGenoTransmitter()] + ops, *args, **kwargs)

def randomMating(numOffspring = 1., sexMode = RandomSex, ops = [], subPopSize = [],
        subPop = (), weight = 0, selectionField = 'fitness'):
    'A basic diploid sexual random mating scheme.'
    return homoMating(
        chooser = randomParentsChooser(True, selectionField),
        generator = mendelianOffspringGenerator(ops, numOffspring, sexMode),
        subPopSize = subPopSize,
        subPop = subPop,
        weight = weight)
```

2.8.2 `homoMating` mating scheme

`homoMating` is used to define all pre-defined homogeneous mating schemes. It takes five parameters: `chooser` (a *parent chooser* that is responsible for choosing one or two parents from the parental generation), `generator` (an *offspring generator* that is responsible for generating a number of offspring from the chosen parents), `subPopSize` (parameter to control offspring subpopulation sizes), `subPop` (applicable subpopulation or virtual subpopulation), 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 2.7.1 for details), parameters `subPop` and `weight` are ignored. Otherwise, the number of offspring this mating scheme will produce is determined by the heterogeneous mating scheme. Figure

Parameters `subPopSize`, `subPop` and `weight` are more or less standard but different parent choosers and offspring generators can be combined to define a large number of homogeneous mating schemes. 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 2.50.

Example 2.50: Define a sequential selfing mating scheme

```
>>> simu = simulator(population(100, loci=[5]*3, infoFields=['parent_idx']),
...     homoMating(sequentialParentChooser(), selfingOffspringGenerator()))
>>> simu.evolve(
...     preOps = [initByFreq([0.2]*5)],
...     ops = [
...         parentTagger(),
...         dumper(structure=False, stage=PrePostMating, max=5)],
...     gen = 1
... )
Subpopulation 0 (unnamed), 100 individuals:
   0: FU 10200 31403 20112 | 13331 02132 40423 |  0
   1: MU 24014 32314 20312 | 34411 11041 22010 |  0
   2: FU 44111 10023 42044 | 00232 02341 23033 |  0
   3: MU 32333 44404 34310 | 02002 11321 02413 |  0
   4: FU 22040 22314 13230 | 01132 22022 44421 |  0
Subpopulation 0 (unnamed), 100 individuals:
   0: MU 10200 02132 40423 | 13331 02132 20112 |  0
   1: FU 34411 32314 20312 | 24014 32314 20312 |  1
   2: FU 44111 10023 23033 | 00232 10023 42044 |  2
   3: MU 32333 11321 34310 | 32333 44404 02413 |  3
   4: MU 01132 22314 44421 | 01132 22314 13230 |  4
(1,)
>>>
```

The simuPOP reference manual lists all pre-defined parent choosers and offspring generators. They may or may not work together depending on the number of parents a parent chooser produces, and the number of parents an offspring generator can handle. You can also define your own parent choosers and offspring generators, as shown below.

2.8.3 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 2.7.2 and 2.7.3.

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. The standard `offspringGenerator` does not have any default operator so no genotype will be transmitted by default. A number of specialized offspring generators are therefore defined. For example, a `mendelianOffspringGenerator` in Example 2.49 uses a `mendelianGenoTransmitter` as the default genotype transmitter. Additional during-mating operators can be added to the operator list, as shown in Example 2.48, but the `mendelianGenoTransmitter` will always be used to transmit genotypes.

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 Peng et al. [2007] for rationals behind such an offspring generator and its applications in the simulation of complex human diseases.

Example 2.51 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 2.51: A controlled random mating scheme

```
>>> def traj(gen):
...     return [0.5 + gen * 0.01]
...
>>> simu = simulator(population(1000, loci=[10]*2),
...     homoMating(randomParentChooser(),
...         controlledOffspringGenerator(loci=[5],
...             alleles=[0], freqFunc=traj,
...             ops = [selfingGenoTransmitter()]))
... )
>>>
>>> # evolve the population while keeping allele frequency 0.5
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [stat(alleleFreq=[5, 15]),
...         pyEval(r'"%.2f\t%.2f\n" % (alleleFreq[5][0], alleleFreq[15][0])')],
...     gen = 5
... )
0.50    0.53
0.51    0.51
0.52    0.52
0.53    0.51
0.54    0.52
(5,)
>>>
>>>
```

2.8.4 Pre-defined genotype transmitters

Although any during mating operators can be used in parameter `ops` of an offspring generator, only those that transmit genotype from parents to offspring are 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.

Genotype transmitters in an offspring generator are the *default* transmitters that can be replaced by another transmitter. If another transmitter is used in the `evolve` function, it will override the default transmitter. For example, a `recombiantor` will override the `mendelianGenoTransmitter()` used in a `randomMating` mating scheme. This is usually not a concern, but will be important if you are defining your own genotype transmitters.

All genotype transmitters only handle known chromosome types such as Autosome, ChromosomeX and ChromosomeY. Customized chromosomes are left untouched because simuPOP does not know how they should be transmitted from parents to offspring. In case that Customized chromosomes are treated as mitochondrial chromosomes, a `mitochondrialGenoTransmitter` can be used to transmit Customized chromosomes randomly from mother to offspring. Example 2.52 demonstrates the use of a `recombinator` to recombine an autosome and two sex chromosomes, and a `mitochondrialGenoTransmitter` to transmit mitochondrial chromosomes. Note that,

- These two operators can be used in the ops parameter of both the `evolve` function and the mating scheme. Recombinator overrides the default Mendelian genotype transmitter defined in the random mating scheme.
- Different applicability rules are applied to these operators when they are used in the `evolve` function or in a mating scheme. Namely, it is possible to apply recombination at certain generations if it is used in the evolve function, and it is possible to apply recombination to individuals in a virtual subpopulation if it is used in the mating scheme.
- Because the mitochondrial genotype transmitter is only applied to customized chromosomes, it is not considered as a genotype transmitter so that it will not override any default genotype transmitter.

Example 2.52: Transmission of mitochondrial chromosomes

```
>>> pop = population(10, loci=[5]*5,
...     # one autosome, two sex chromosomes, and two mitochondrial chromosomes
...     chromTypes=[Autosome, ChromosomeX, ChromosomeY] + [Customized]*2,
...     infoFields=['father_idx', 'mother_idx'])
>>>
>>> simu = simulator(pop, randomMating(ops=[mitochondrialGenoTransmitter()]))
>>>
>>> simu.evolve(
...     preOps=[initByFreq([0.4] + [0.2]*3)],
...     ops=[
...         recombinator(rate=0.1),
...         parentsTagger(),
...         dumper(structure=False),
...     ],
...     gen = 2
... )
Subpopulation 0 (unnamed), 10 individuals:
   0: MU 03222 21102 _____ 11002 20203 | 11201 _____ 21330 00000 00000 |  1 6
   1: FU 32002 02001 _____ 32023 32023 | 00212 30221 _____ 00000 00000 |  9 0
   2: FU 10112 00032 _____ 30100 20220 | 20312 03013 _____ 00000 00000 |  8 7
   3: FU 21213 21222 _____ 33033 33033 | 21000 10130 _____ 00000 00000 |  2 4
   4: MU 10112 00032 _____ 30100 20220 | 23331 _____ 00133 00000 00000 |  2 7
   5: MU 22002 01201 _____ 00033 00033 | 03010 _____ 23132 00000 00000 |  3 0
   6: FU 22130 02010 _____ 00033 10033 | 31030 23010 _____ 00000 00000 |  1 5
   7: FU 22130 02010 _____ 00033 00033 | 31201 23010 _____ 00000 00000 |  1 5
   8: MU 10112 33020 _____ 20220 20220 | 20010 _____ 23132 00000 00000 |  3 7
   9: MU 10220 00032 _____ 30100 20220 | 23330 _____ 00133 00000 00000 |  2 7
Subpopulation 0 (unnamed), 10 individuals:
   0: FU 20312 03013 _____ 30100 30100 | 03222 21102 _____ 00000 00000 |  0 2
   1: MU 22130 02010 _____ 00033 00033 | 02002 _____ 23132 00000 00000 |  5 7
   2: FU 20312 00032 _____ 30100 20220 | 11201 21102 _____ 00000 00000 |  0 2
   3: MU 21213 21230 _____ 33033 33033 | 23330 _____ 00133 00000 00000 |  9 3
   4: MU 31030 02010 _____ 10033 10033 | 10312 _____ 00133 00000 00000 |  4 6
   5: FU 31201 02010 _____ 00033 00033 | 10112 33020 _____ 00000 00000 |  8 7
   6: FU 21000 10130 _____ 33033 33033 | 23331 00032 _____ 00000 00000 |  4 3
   7: MU 10112 03032 _____ 20220 20220 | 03221 _____ 21330 00000 00000 |  0 2
   8: MU 21200 10132 _____ 33033 33033 | 10112 _____ 23132 00000 00000 |  8 3
   9: MU 31230 02010 _____ 00033 00033 | 23331 _____ 00133 00000 00000 |  4 7
(2,)
>>>
```

2.8.5 Customized genotype transmitter

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 2.30), it is much better to derive an operator from pyOperator, as the case in Example 2.33.

Example 2.54 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`. This operator will override the default Mendelian genotype transmitter used in the random mating scheme because `formOffGenotype=True`.

The actual function to transmit parental genotype is `self.transmitGenotype`. This function initializes two recombinators if they have not been initialized and uses them to transmit parental genotypes. Example 2.53 outputs the population in two generations. You should notice that paternal chromosome are not recombined when they are transmitted to offspring.

Example 2.53: A customized genotype transmitter for sex-specific recombination

```
>>> class sexSpecificRecombinator(pyOperator):
...     def __init__(self, intensity=0, rate=0, loci=[], convMode=NoConversion,
...             maleIntensity=0, maleRate=0, maleLoci=[], maleConvMode=NoConversion,
...             *args, **kwargs):
...         # This operator is used to recombine maternal chromosomes
...         self.recombinator = recombinator(intensity, rate, loci, convMode)
...         # This operator is used to recombine paternal chromosomes
...         self.maleRecombinator = recombinator(maleIntensity, maleRate,
...             maleLoci, maleConvMode)
...         #
...         self.initialized = False
...         # Note the use of parameter formOffGenotype.
...         pyOperator.__init__(self, func=self.transmitGenotype,
...             stage=DuringMating, formOffGenotype=True, *args, **kwargs)
...     #
...     def transmitGenotype(self, pop, off, dad, mom):
...         # Recombinators need to be initialized. Basically, they cache some
...         # population properties to speed up genotype transmission.
...         if not self.initialized:
...             self.recombinator.initialize(pop)
...             self.maleRecombinator.initialize(pop)
...             self.initialized = True
...         # Form the first homologous copy of offspring.
...         self.recombinator.transmitGenotype(mom, off, 0)
...         self.maleRecombinator.transmitGenotype(dad, off, 1)
...         return True
...
>>> pop = population(10, loci=[15]*2, infoFields=['father_idx', 'mother_idx'])
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps=[initByFreq([0.4] + [0.2]*3)],
...     ops=[
...         parentsTagger(),
...         sexSpecificRecombinator(rate=0.1, maleRate=0),
...         dumper(structure=False),
...     ],
...     gen = 2
... )
Subpopulation 0 (unnamed), 10 individuals:
   0: FU 222210022121010 233100130022313 | 203101202100003 010110332030000 |  7 9
   1: MU 103210022121001 233100032212313 | 211231000221011 010202100300312 |  1 9
   2: MU 003202220010002 103113113113000 | 100120020003003 022002013331312 |  4 0
   3: FU 032223122121101 013100220000002 | 211231000221011 010202100300312 |  1 0
   4: MU 123220210302313 023230303211233 | 001121321302303 232010011301000 |  6 5
   5: FU 303202220010001 013100220000100 | 203101202100003 010110332030000 |  7 0
   6: MU 123220003012213 023230303211233 | 020001300202300 213202301203001 |  2 5
   7: FU 003202222121002 103100223113122 | 110010032032312 310101021003103 |  8 0
   8: MU 003202220111102 103110220000122 | 230310300123111 310101021003103 |  8 0
```

```
   9: MU 133010003012213 023230303211233 | 033213301321023 232010011301000 |  6 5
Subpopulation 0 (unnamed), 10 individuals:
   0: MU 110202222121002 310101021013122 | 001121321302303 232010011301000 |  4 7
   1: MU 203101202100003 233110332030013 | 001121321302303 023230303211233 |  4 0
   2: FU 010010032032312 103100223113122 | 020001300202300 023230303211233 |  6 7
   3: MU 222210022121010 033100130030000 | 020001300202300 213202301203001 |  6 0
   4: FU 203102222010001 010110332030000 | 001121321302303 023230303211233 |  4 5
   5: MU 110010022121002 310101021003103 | 123220210302313 232010011301000 |  4 7
   6: FU 031223000221011 010202220300312 | 033213301321023 232010011301000 |  9 3
   7: MU 032223000221111 013100220000002 | 123220003012213 213202301203001 |  6 3
   8: MU 303102220000001 013100220000100 | 103210022121001 233100032212313 |  1 5
   9: MU 032231022121011 010202100300002 | 211231000221011 010202100300312 |  1 3
(2,)
>>>
```

2.8.6 Pre-define parent choosers

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 natual selection. Please refer to the simuPOP reference manual for details about these objects.

We have seen sequential and random parent choosers in Examples 2.50 and 2.51. A less-used parent chooser is `infoParentsChooser`, which chooses a parent randomly, and his/her spouse from indexes stored in his/her information fields. This parent chooser is usually used in a consanguineous mating scheme where certain types of relatives of each individual are stored in his/her information fields, and used during the selection of spouses. For example, a `consanguineousMating` mating scheme in Example 2.54 produces a male and a female offspring at each mating event. Before mating, a function is called to record every individual's sibling in his/her information field `sibling`. During mating, a parent is chosen randomly, and mates with his/her sibling.

Example 2.54: A consanguineous mating scheme.

```
>>> pop = population(100, loci=[10],
...     infoFields=['father_idx', 'mother_idx', 'sibling'])
>>>
>>> pop.setVirtualSplitter(sexSplitter())
>>>
>>> def locate_sibling(pop):
...     '''The population is arranged as MFMFMFMF... where MF are siblings, so the
...     sibling of males are 1, 3, 5, .. and the slibling of females are 0, 2, 4, ...
...     '''
...     pop.setIndInfo([2*x+1 for x in range(pop.popSize()/2)], 'sibling', (0, 0))
...     pop.setIndInfo([2*x for x in range(pop.popSize()/2)], 'sibling', (0, 1))
...
>>> simu = simulator(pop, consanguineousMating(func=locate_sibling, infoFields=['sibling'],
...     numOffspring=2, sexMode=(NumOfMale, 1)))
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8], sex=[Male, Female])],
...     ops = [
...         parentsTagger(),
...         dumper(structure=False, max=6, at=[-1])
...     ],
...     gen = 2
... )
Subpopulation 0 (unnamed), 100 individuals:
   0: MU 1111111111 | 1111111111 |  62 63 0
   1: FU 0111111111 | 1111101101 |  62 63 0
   2: MU 1111111011 | 1100111110 |  50 51 0
```

```
   3: FU 1111111011 | 1111111011 |  50 51 0
   4: MU 1111111011 | 1111111111 |  88 89 0
   5: FU 1111111011 | 1110110111 |  88 89 0
(2,)
>>>
```

This example does not make such sense because consanguineous mating usually happens between first and second degree relatives, and represents only a small fraction of total parents in a population. `doc/cook/Mating_consanguineous.py` gives a more realistic example in which certain proportion of offspring are produced by random mating, and others are results of marriages between first-degree cousins.

2.8.7 A Python parent chooser

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 2.55 defines a function that calculate $f(k) = \sum_{i=1}^{k} \frac{1}{i}$ for $k = 1, ..., 5$. It does not calculate each $f(k)$ repeatedly but returns $f(1)$, $f(2)$, ... sequentially.

Example 2.55: A sample generator function

```
>>> def func():
...     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
>>>
```

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 (*both are 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 2.56 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 2.56: A hybrid parent chooser that chooses parents by their social status

```
>>> from random import randint
>>> def randomChooser(pop, sp):
...     males = []
...     females = []
```

```
...     # identify males and females in each social rank
...     for rank in range(3):
...         males.append([x for x in range(pop.subPopSize(sp)) \
...             if pop.individual(x, sp).sex() == Male and \
...                 pop.individual(x, sp).info('rank') == rank])
...         females.append([x for x in range(pop.subPopSize(sp)) \
...             if pop.individual(x, sp).sex() == Female and \
...                 pop.individual(x, sp).info('rank') == rank])
...     while True:
...         # choose a parent randomly
...         idx = randint(0, pop.subPopSize(sp) - 1)
...         par = pop.individual(idx, sp)
...         # then choose a spouse in the same rank randomly
...         if par.sex() == Male:
...             rank = par.intInfo('rank')
...             yield idx, females[rank][randint(0, len(females[rank]) - 1)]
...         else:
...             rank = par.intInfo('rank')
...             yield males[rank][randint(0, len(males[rank]) - 1)], idx
...
>>> def setRank(pop, dad, mom, off):
...     'The rank of offspring can increase or drop to zero randomly'
...     off.setInfo((dad.info('rank') + randint(-1, 1)) % 3, 'rank')
...
>>> pop = population(size=[1000, 2000], loci=[1], infoFields=['rank'])
>>> pop.setIndInfo([randint(0, 2) for x in range(pop.popSize())], 'rank')
>>>
>>> simu = simulator(pop, homoMating(
...     pyParentsChooser(randomChooser),
...     mendelianOffspringGenerator()))
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [],
...     gen = 5
... )
(5,)
>>>
>>>
```

2.8.8 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 vincinity, 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 2.57 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 2.57: 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));
        }

        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;
};
```

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 2.58 lists an interface file for the C++ class defined in Example 2.57. Please refer to the SWIG reference manual for details.

Example 2.58: 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"
```

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 `distutil` module of Python handle all the details for you. A typical `setup.py` file is demonstrated in Example 2.59.

Example 2.59: 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', '-templatereduce', '-shadow',
                '-python', '-c++', '-keyword', '-nodefaultctor'],
            include_dirs = ["."],
    )
  ]
)
```

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 2.57 demonstrates how to use such a C++ parents chooser in your simuPOP script. It uses the same Python parent chooser interface as in 2.56, but leaves all the (potentially) computationally intensive parts to the C++ level `myParentsChooser` object.

Example 2.60: Implement a parent chooser in C++

```
# The class myParentsChooser is defined in module myParentsChooser
from myParentsChooser import myParentsChooser

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() == Male],
        [x for x in range(pop.popSize()) if pop.individual(x).sex() == Female])
    while True:
        # return indexes of parents repeatedly
        yield pc.chooseParents()

pop = population(100, loci=[1])
simu = simulator(pop,
    homoMating(pyParentsChooser(parentsChooser), mendelianOffspringGenerator())
)
simu.evolve(
    preOps = [initByFreq([0.5, 0.5])],
    ops = [],
    gen = 100
)
```

2.8.9 The pedigree mating scheme

This feature is still under major revision.

2.9 Simulator

A simuPOP simulator evolves one or more copies of a population forward in time, subject to various operators. Although simulators have been used extensively in the previous chapters, it is worthwhile to have a detailed look at this object.

2.9.1 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 simulator has generation number 0, a simulator would be at the beginning of generation n after it evolves n generations. The generation number would increase if the simulator continues to evolve. During evoluting, 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 2.61 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 `evolve` function twice so the second part starts at generation 5. You can also use `simu.setGen(0)` to reset the generation number if you would like to have a fresh start for the second `evove()` call.

Example 2.61: Generation number of a simulator

```
>>> simu = simulator(population(50, loci=[10], ploidy=1),
...     randomSelection(), rep=3)
>>> simu.evolve(ops = [], gen = 5)
(5, 5, 5)
>>> simu.gen()
5
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [
...         stat(alleleFreq=[5]),
...         ifElse('alleleNum[5][0] == 0',
...             pyEval(r"'Allele 0 is lost in rep %d at gen %d\n' % (rep, gen)")),
...         ifElse('alleleNum[5][0] == 50',
...             pyEval(r"'Allele 0 is fixed in rep %d at gen %d\n' % (rep, gen)")),
...         terminateIf('alleleNum[5][0] == 0 or alleleNum[5][0] == 50'),
...     ],
... )
Allele 0 is lost in rep 0 at gen 70
Allele 0 is fixed in rep 2 at gen 76
Allele 0 is lost in rep 1 at gen 140
(66, 136, 72)
>>> simu.gen()
141
>>>
```

2.9.2 Operator calling sequence

Operators can be applied at different stages of a life cycle (pre-, during-, and post-mating, controlled by parameter `stage`), at specified generations (controlled by parameters `begin, end, step` and `at`), and to specified replicates (controlled by parameter `rep`). The order at which operators are applied is usually clear but can become confusing when the number of operators increases. For example, `stat(...)` should be put before any operator (such as an terminator) that uses the shared variable set by this operator. An error will occur if the variables are used before they are set.

Because it is not always clear which stage(s) an operator can be applied and in which order they will be applied, a parameter `dryrun` is provided to the `simulator::evolve()` function. If set to `True`, the `evolve` function will list all operators in the order at which they will be applied. Example 2.62 shows the operator calling sequence for Example 2.26.

Example 2.62: List the order at which operators are applied

```
>>> simu = simulator(population(100, loci=[20]), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [
...         stat(alleleFreq=[0], begin=80, step=10),
...         pyEval(r"'After gen %d: allele freq: %.2f\n' % (gen, alleleFreq[0][0])",
...             begin=80, step=10),
...         pyEval(r"'Around gen %d: alleleFreq: %.2f\n' % (gen, alleleFreq[0][0])",
...             at = [-10, -1], stage=PrePostMating)
...     ],
...     postOps = [savePopulation(output='sample.pop')],
...     gen=100,
...     dryrun = True
... )
Dryrun mode: display calling sequence
Apply pre-evolution operators
  Replicate 0
      - <simuPOP::initByFreq> end at 1
Start evolution
  Replicate 0
    Pre-mating operators
      - <simuPOP::pyEval> at generation(s) -10 -1
    Start mating
    Apply post-mating operators
      - <simuPOP::statistics> begin at 80 at interval 10
      - <simuPOP::pyEval> begin at 80 at interval 10
      - <simuPOP::pyEval> at generation(s) -10 -1
Apply post-evolution operators:
  Replicate 0
      - <simuPOP::save population> at all generations
(0,)
>>>
```

2.9.3 Population access and other simulator operations

Function `population()` and `populations()` are provided to access populations within a simulator. Similar to functions `individual()` and `individuals()` for a population, `population(rep)` returns a reference to the `rep`th population in a simulator and `populations()` returns an Python iterator that can be used to iterate through all populations. Modifying these references will change the corresponding populations within the simulator. An independent copy of a population can be made using the `clone()` function of a population (e.g. `simu.population(0).clone()`).

Populations in a simulator can be added or removed using functions `add()` and `extract()`. The **newly added populations do not have to have the same genotypic structure as existing populations**. However, because the same operators will be applied to all populations, it is your responsibility to make sure that the operators can be applied to these populations.

Just like populations, a simulator can be cloned, saved and loaded. This makes it easy to stop a simulator, take a snapshot and resume evolution. It is even easy to save a simlator, transfer it to another machine and resume the evolution over there. Because **virtual splitters are not saved with populations**, you will have to re-assign splitters to populations if they are needed for subsequent simulations.

Example 2.63: Clone, save and load a simulator

```
>>> simu = simulator(population(100, loci=[5, 10], infoFields=['x']),
...     randomMating(), rep=5)
>>> simu.evolve(preOps=[initByFreq([0.4, 0.6])],
...     ops=[], gen=10)
(10, 10, 10, 10, 10)
>>> # clone
>>> cloned = simu.clone()
>>> # save and load, using a different mating scheme
>>> simu.save("sample.sim")
>>> loaded = LoadSimulator("sample.sim", randomMating(numOffspring=2))
>>> #
>>> simu.numRep()
5
>>> loaded.numRep()
5
>>> for pop1,pop2 in zip(cloned.populations(), loaded.populations()):
...     assert pop1 == pop2
...
>>> # continue to evolve
>>> simu.evolve(ops=[], gen=10)
(10, 10, 10, 10, 10)
>>> simu.gen()
20
>>>
```

2.9.4 Modifying populations and mating scheme

Although a standard Wright-Fisher random mating scheme is usually preferred because it leads to a larger effective population size than other mating schemes, it is difficult to ascertain pedigrees from a random mating population because there will be very few siblings in such a population. In addition, because we usually only sample from the last few generations, it would be more efficient to keep tract of pedigree information only for these generations. Such considerations lead to the popularity of a two stage evolutionary scenario where the standard random mating scheme is used in the first stage and another mating scheme that is more suitable for pedigree ascertainment is used in the second stage. Example 2.64 demonstrates the implementation of such a scenario.

Example 2.64: A two-stage evolutionary process

```
>>> # First stage: use the standard random mating scheme, do not use any
>>> # information field for efficiency considerations.
>>> simu = simulator(population(500, loci=[10]), randomMating())
>>> simu.evolve(preOps = [initByFreq([0.5, 0.5])],
...     ops = [], gen = 50)
(50,)
>>> # Second stage: track parents and produce more offspring per mating
>>> # event. In preparation for pedigree ascertainment.
>>> for pop in simu.populations():
```

```
...     pop.addInfoFields(['father_idx', 'mother_idx'])
...     pop.setAncestralDepth(1)
...
>>> simu.setMatingScheme(randomMating(numOffspring=2))
>>> simu.evolve(
...     ops = [
...         parentsTagger(),
...         maPenetrance(loci=0, penetrance=(0.2, 0.4, 0.5))
...     ],
...     gen = 5
... )
(5,)
>>> # Sample affected sibpairs
>>> pop = simu.extract(0)
>>> sample = AffectedSibpairSample(pop, size=5)[0]
>>> [ind.intInfo('father_idx') for ind in sample.individuals()]
[44, 44, 287, 287, 431, 431, 309, 309, 451, 451]
>>>
```

2.9.5 Change genotypic structure during evolution

Most operators do not change the genotypic structure of populations during evolution. However, it is possible to change the structure of a population, such as adding or removing information fields, loci or chromosomes during evolution. The only restriction is that all individual in a population needs to have the same genotypic structure. That is to say, if you are inserting a new locus to an individual, all individuals in this population should have it. This is why there is no individual-level structure-modification functions.

Example 2.65 gives an example of a dynamic mutator. This mutator is not a conventional mutator in that it does not mutate any existing loci. It assumes a chromosome region that originally has no polymorphic markers. When a mutation happens, a monomorphic marker that is not simulated becomes polymorphic and is inserted to the chromosome. If the region is long enough, this example effectively simulates an infinite allele mode.

Example 2.65: A Python mutator that adds new loci to populations.

```
>>> import random
>>> def mutator(pop, param):
...     'Parameter has a length of region and a mutation rate at each basepair'
...     region, rate = param
...     # there are certainly more efficient algorithm, but this
...     # example just mutate each basepair one by one....
...     for i in range(region):
...         if random.random() < rate:
...             try:
...                 idx = pop.addLoci(chrom=0, pos=i)[0]
...             except:
...                 # position might duplicate
...                 continue
...             # choose someone to mutate
...             ind = pop.individual(random.randint(0, pop.popSize() - 1))
...             ind.setAllele(1, idx)
...     return True
...
>>> # The populations start with no loci at all.
>>> simu = simulator(population(1000, loci=[]), randomMating(), rep=3)
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [pyOperator(func=mutator, param=(10000, 2e-6))],
...     gen = 200
```

```
... )
(200, 200, 200)
>>> for pop in simu.populations():
...     print pop.totNumLoci(), pop.lociPos()
...
3 (3512.0, 3991.0, 7154.0)
1 (8655.0,)
2 (6186.0, 6321.0)
>>>
```

2.10 Pedigrees

This feature is still under major revision.

Chapter 3

simuPOP Operators

simuPOP is large, consisting of more than 80 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 D, D' and r^2 linkage disequilibrium measures; F_{st}, F_{it} and F_{is}); 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.

3.1 Initialization

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

3.1.1 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. In the second case, a sequence of sex (`Male` or `Female`) is assigned to individuals succesively. 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 3.1 uses two `initSex` operators to initialize two subpopulations.

Example 3.1: Initialize individual sex

```
>>> pop = population(size=[1000, 1000])
>>> InitSex(pop, maleFreq=0.3, subPops=0)
>>> InitSex(pop, sex=[Male, Female, Female], subPops=1)
>>> Stat(pop, numOfMale=True)
>>> print pop.dvars(0).numOfMale
316
>>> print pop.dvars(1).numOfMale
334
>>>
```

3.1.2 Initialize by allele frequency (operator `initByFreq`)

Operator `initByFreq` (and its function form `InitByFreq`) initialize individual genotype by **allelic spectrum**, which is the number and frequency of alleles at a locus. For example, `alleleFreq=(0, 0.2, 0.4, 0.2)` will yield allele 0, 1, 2, and 3 with probability 0, 0.2, 0.4 and 0.2 respectively. 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. In the latter case, a list of allelic spectra can be given to assign different genotype with different allele frequency for each (virtual) subpopulation.

Example 3.2: Initialize by allele frequency

```
>>> pop = population(size=[2, 3], loci=[5, 7])
>>> InitByFreq(pop, alleleFreq=[[.2, .8], [.8, .2]])
>>> Dump(pop, structure=False)
Subpopulation 0 (unnamed), 2 individuals:
   0: FU 10111 0001110 | 11111 1101111
   1: MU 11110 1110110 | 01111 1001111
Subpopulation 1 (unnamed), 3 individuals:
   2: MU 00000 0000101 | 00110 0010000
   3: FU 00000 0000000 | 00011 0000000
   4: FU 00000 0010100 | 00000 0010000
>>>
```

It is sometimes desired to creates identical individuals with random genotype. Parameter `identicalInds` can be used for this purpose. When this parameter is set to true, a random individual will be created for each subpopulation (using different allele frequencies if a list of allelic spectra are given), and be copied to all other individuals in the subpopulation. Example 3.3 demonstrates this usage.

Example 3.3: Initialize by allele frequency with identical individuals in each subpopulation

```
>>> pop = population(size=[2, 3], loci=[5, 7])
>>> InitByFreq(pop, alleleFreq=[.2, .8], identicalInds=True)
>>> Dump(pop, structure=False)
Subpopulation 0 (unnamed), 2 individuals:
   0: FU 11100 1111111 | 11101 1111011
   1: FU 11100 1111111 | 11101 1111011
Subpopulation 1 (unnamed), 3 individuals:
   2: MU 00110 1110111 | 11111 0111011
   3: FU 00110 1110111 | 11111 0111011
   4: FU 00110 1110111 | 11111 0111011
>>>
```

For convenience and for backward-compatibility, this operator by default also initialize individual sex (parameters *maleFreq* and *sex* are accepted). If this is not needed (e.g. indiviudal sex has already been initialized), you can set paramter *initSex* to `False`.

3.1.3 Initialize by haplotype (operator `initByValue`)

Operator `initByValue` (and its function form `InitByValue`) initializes individual genotypes using given haplotypes. The simplest form of this operator is to specify genotype on one or all homologous sets of chromosomes. For example, all individuals in Example 3.4 get the same genotype using such an operator.

Example 3.4: initialize by haplotype

```
>>> pop = population(size=[2, 3], loci=[5, 7])
>>> InitByValue(pop, [1]*5 + [2]*7 + [3]*5 +[4]*7)
>>> Dump(pop, structure=False)
Subpopulation 0 (unnamed), 2 individuals:
   0: MU 11111 2222222 | 33333 4444444
```

```
   1: MU 11111 2222222 | 33333 4444444
Subpopulation 1 (unnamed), 3 individuals:
   2: FU 11111 2222222 | 33333 4444444
   3: MU 11111 2222222 | 33333 4444444
   4: FU 11111 2222222 | 33333 4444444
>>>
```

A number of parameters are provided to initialize individual genotype at a finer scale. More specifically, you can apply the operator to specified loci (parameter *loci*), (virtual) subpopulations (parameter `subPops`), homologous sets of chromosomes (parameter `ploidy`). If multiple haplotypes are given, you can specify the probabilities at which each haplotype will be used using parameter `proportions`. Example 3.5 demonstrates the use of these parameters. Note that operator `initByValue` also initializes individual sex so *`initSex`* should be set to `False` when multiple initializers are applied.

Example 3.5: initialize by haplotypes with given proportion

```
>>> pop = population(size=[6, 8], loci=[5, 7])
>>> pop.setVirtualSplitter(sexSplitter())
>>> # initialize sex and the first two loci
>>> InitByValue(pop, loci=range(5), value=range(10))
>>> # initialize all males
>>> InitByValue(pop, loci=range(5, 12), value=[2]*7,
...     subPops=[(0, 0), (1, 0)], initSex=False)
>>> # initialize females by proportion
>>> InitByValue(pop, loci=range(5, 12), ploidy=1, value=[[3]*7, [4]*7],
...     initSex=False, subPops=[(0, 1), (1, 1)], proportions=[0.4, 0.6])
>>> Dump(pop, structure=False)
Subpopulation 0 (unnamed), 6 individuals:
   0: MU 01234 2222222 | 56789 2222222
   1: FU 01234 0000000 | 56789 3333333
   2: MU 01234 2222222 | 56789 2222222
   3: FU 01234 0000000 | 56789 3333333
   4: MU 01234 2222222 | 56789 2222222
   5: MU 01234 2222222 | 56789 2222222
Subpopulation 1 (unnamed), 8 individuals:
   6: FU 01234 0000000 | 56789 3333333
   7: MU 01234 2222222 | 56789 2222222
   8: MU 01234 2222222 | 56789 2222222
   9: MU 01234 2222222 | 56789 2222222
  10: FU 01234 0000000 | 56789 4444444
  11: FU 01234 0000000 | 56789 3333333
  12: FU 01234 0000000 | 56789 4444444
  13: MU 01234 2222222 | 56789 2222222
>>>
```

3.2 Expressions and statements

3.2.1 Output an 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)`.

3.2.2 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 3.6 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 calcuated (operator `stat`) and appended to this variable (operator `pyExec`). The result is a trajectory of allele frequencies during evolution.

Example 3.6: Execute Python statements during evolution

```
>>> simu = simulator(population(100, loci=[1]),
...     randomMating(), rep=2)
>>> simu.evolve(
...     preOps = [
...         initByFreq([0.2, 0.8]),
...         pyExec('traj=[]')
...     ],
...     ops = [
...         stat(alleleFreq=[0]),
...         pyExec('traj.append(alleleFreq[0][1])'),
...     ],
...     gen=5
... )
(5, 5)
>>> # print trajectory
>>> print ', '.join(['%.3f' % x for x in simu.dvars(0).traj])
0.765, 0.800, 0.805, 0.825, 0.800
>>>
```

3.2.3 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 1.1, 2.20, 2.26 and 2.28) 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 3.7 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 alone with a generation number.

Example 3.7: Evaluate a expression and statements in a population's local namespace.

```
>>> simu = simulator(population(1000, loci=[1],
...     infoFields=['mother_idx', 'father_idx']),
...     randomMating())
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         stat(alleleFreq=[0]),
...         parentsTagger(),
...         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 416, #mother 430
gen 1, #father 422, #mother 441
gen 2, #father 443, #mother 429
(3,)
>>>
```

Note that the function form of this operator (`PyEval`) returns the result of the expression rather than writting it to an output.

3.2.4 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 contraray, work at the individual level, using individual information fields.

Because there is no individual-specific namespace, these two operators make use of either a temporary namespace for every individual, or the population namespace (parameter *usePopVars*). In the first case, a namespace is created for each invidividual, with variables being the information fields of this individual. In the second case, individual information fields are copied to the population namespace one by one. Expressions and statements can make use of population variables in this case. Optionally, the individual object can be exposed to these namespace using a user-specified name (parameter *exposeInd*).

Operator `infoEval` evaluates an expression and outputs its value. Operator `infoExec` executes one or more statements and does not produce any output. The major difference between them is that `infoEval` does not change individual information fields while `infoExec` update individual information fields from the namespace after the statements are executed.

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 $a + b^2$ for information fields a and b for all individuals.

Example 3.8 demonstrates the use of this operator.

Example 3.8: Evaluate expressions using individual information fields

```
>>> import random
>>> pop = population(20, loci=[1], infoFields=['a'])
>>> pop.setVirtualSplitter(infoSplitter('a', cutoff=[3]))
>>> InitByFreq(pop, [0.2, 0.8])
>>> pop.setIndInfo([random.randint(2, 5) for x in range(20)], 'a')
>>> InfoEval(pop, 'a', subPops=[(0, 0)]);print
2.02.02.02.02.0
>>> InfoEval(pop, 'ind.allele(0, 0)', exposeInd='ind');print
11000011110111110100
>>> # use population variables
>>> pop.dvars().b = 5
```

```
>>> InfoEval(pop, '"%d " % (a+b)', usePopVars=True);print
10 8 7 10 7 10 8 9 10 8 8 10 9 10 7 9 7 7 8 10
>>>
```

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 3.9 demonstrates the use of this operator, using its function form `InfoExec`.

Example 3.9: Execute statements using individual information fields

```
>>> pop = population(100, loci=[1], infoFields=['a', 'b', 'c'])
>>> InitByFreq(pop, [0.2, 0.8])
>>> 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)
>>> InfoExec(pop, 'b=ind.sex()', exposeInd='ind')
>>> print pop.indInfo('b')[:10]
(1.0, 2.0, 1.0, 2.0, 2.0, 2.0, 1.0, 1.0, 2.0, 2.0)
>>> InfoExec(pop, 'c=a+b')
>>> print pop.indInfo('c')[:10]
(2.0, 3.0, 2.0, 3.0, 3.0, 3.0, 2.0, 2.0, 3.0, 3.0)
>>> pop.dvars().d = 5
>>> InfoExec(pop, 'c+=d', usePopVars=True)
>>> print pop.indInfo('c')[:10]
(7.0, 8.0, 7.0, 8.0, 8.0, 8.0, 7.0, 7.0, 8.0, 8.0)
>>>
```

Note: Except for the local (temporary) namespace, operator `infoEval` and `infoExec` can also access variables and functions in a global namespace, which is the module namespace of your script. However, use of global variables in these operators are strongly discouraged.

3.3 Demographic changes

A mating scheme controls the size of an offspring generation using parameter `subPopSize`. This parameter has been described in detail in section 2.7.1. 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).

3.3.1 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 m by n matrix (a nested list in Python) that specifies migration rate from each source to each destination subpopulation. m and n are determined by the number of *from* and *to* subpopulations.

Example 3.10 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 befor migration.
- Migration rates to subpopulation themselves are determined automatically so they can be left unspecified.

Example 3.10: Migration by probability

```
>>> simu = simulator(
...     population(size=[1000]*3, infoFields=['migrate_to']),
...     randomMating())
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         migrator(rate=[
...             [0, 0.1, 0.1],
...             [0, 0, 0.1],
...             [0, 0.1, 0]
...         ]),
...         stat(popSize=True),
...         pyEval('subPopSize'),
...         pyOutput('\n')
...     ],
...     gen = 5
... )
[807, 1092, 1101]
[649, 1146, 1205]
[520, 1195, 1285]
[428, 1223, 1349]
[352, 1264, 1384]
(5,)
>>>
```

Migration by proportion and counts

Migration rate specified in the rate parameter in Example 3.10 is intepreted as probabilities. That is to say, a migration rate $r_{m,n}$ is interpreted as the probability at which any individual in subpopulation m migrates to subpopulation n. 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 `ByProportion` or `ByCounts`. The `ByProportion` mode interpret $r_{m,n}$ as proportion of individuals who will migrate from subpopulation m to n so the number of $m \to n$ migrant will be exactly $r_{m,n} \times$subPopSize(m). In the `ByCounts` mode, $r_{m,n}$ is interpretted as number of migrants, regardless the size of subpopulation m. Example 3.11 demonstrates these two migration modes, as well as the use of parameters `subPops` and `toSubPops`.

Example 3.11: Migration by proportion and count

```
>>> simu = simulator(
...     population(size=[1000]*3, infoFields=['migrate_to']),
...     randomMating())
>>> simu.evolve(
...     preOps = [initSex()],
...     ops = [
...         migrator(rate=[[0.1], [0.2]],
...             mode=ByProportion,
...             subPops=[1, 2],
...             toSubPops=[3]),
...         stat(popSize=True),
...         pyEval('subPopSize'),
...         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,)
>>> #
>>> simu.evolve(
...     ops = [
...         migrator(rate=[[50, 50], [100, 50]],
...             mode=ByCounts,
...             subPops=[3, 2],
...             toSubPops=[2, 1]),
...         stat(popSize=True),
...         pyEval('subPopSize'),
...         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,)
>>>
```

Theoretical migration models

To facilitate the use of widely used theoretical migration models, two functions are defined in module `simuUtil`.

- `MigrIslandRates(r, n)` returns a $n \times n$ migration matrix

$$\begin{pmatrix} 1-r & \frac{r}{n-1} & \dots & \dots & \frac{r}{n-1} \\ \frac{r}{n-1} & 1-r & \dots & \dots & \frac{r}{n-1} \\ & & \dots & & \\ \frac{r}{n-1} & \dots & \dots & 1-r & \frac{r}{n-1} \\ \frac{r}{n-1} & \dots & \dots & \frac{r}{n-1} & 1-r \end{pmatrix}$$

- `MigrSteppingStoneRates(r, n, circular=False)` returns a $n \times n$ migration matrix

$$\begin{pmatrix} 1-r & r & & & \\ r/2 & 1-r & r/2 & & \\ & & \dots & & \\ & & r/2 & 1-r & r/2 \\ & & & r & 1-r \end{pmatrix}$$

  and if `circular=True`, returns

$$\begin{pmatrix} 1-r & r/2 & & & r/2 \\ r/2 & 1-r & r/2 & & \\ & & \dots & & \\ & & r/2 & 1-r & r/2 \\ r/2 & & & r/2 & 1-r \end{pmatrix}$$

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. Example 3.12 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 3.12: Migration from virtual subpopulations

```
>>> pop = population(size=[1000]*2, infoFields=['migrate_to'])
>>> pop.setVirtualSplitter(sexSplitter())
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
```

```
...     # 500 males and 500 females
...     preOps = [initSex(sex=[Male, Female])],
...     ops = [
...         migrator(rate=[
...             [0, 0.10],
...             [0, 0.05],
...             ],
...             mode = ByProportion,
...             subPops=[(0, 0), (0, 1)]),
...         stat(popSize=True, numOfMale=True, stage=PrePostMating),
...         pyEval(r"'%d/%d\t%d/%d\n' % (subPop[0]['numOfMale'], subPopSize[0], "
...             "subPop[1]['numOfMale'], subPopSize[1])", stage=PrePostMating),
...     ],
...     gen = 2
... )
450/925 550/1075
445/925 536/1075
401/857 580/1143
427/857 537/1143
(2,)
>>>
```

Arbitrary migration models

If none of the described migration mothods 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=ByIndInfo`. Example 3.13 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 3.13: Manual migration

```
>>> pop = population([10]*2, infoFields=['migrate_to'])
>>> pop.setIndInfo([0, 1, 2, 3]*5, 'migrate_to')
>>> Migrate(pop, mode=ByIndInfo)
>>> pop.subPopSizes()
(5, 5, 5, 5)
>>>
```

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.

3.3.2 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 3.14 is split into three subpopulations of sizes 300, 300 and 400 respectively, after evolving as a single population for two generations.

Example 3.14: Split subpopulations by size

```
>>> simu = simulator(population(1000), randomSelection())
>>> simu.evolve(
```

```
...     ops = [
...         splitSubPops(subPops=0, sizes=[300, 300, 400], at=2),
...         stat(popSize=True),
...         pyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     gen = 4
... )
Gen 0:  [1000]
Gen 1:  [1000]
Gen 2:  [300, 300, 400]
Gen 3:  [300, 300, 400]
(4,)
>>>
```

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, naming apply a `splitSubPops` operator at the right generation. Example 3.15 demonstrates such an evolutionary scenario.

Example 3.15: Split subpopulations by proportion

```
>>> def demo(gen, oldSize=[]):
...     if gen < 2:
...         return 1000 + 100 * gen
...     else:
...         return [x + 50 * gen for x in oldSize]
...
>>> simu = simulator(population(1000),
...     randomSelection(subPopSize=demo))
>>> simu.evolve(
...     ops = [
...         splitSubPops(subPops=0, proportions=[.5]*2, at=2),
...         stat(popSize=True),
...         pyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     gen = 4
... )
Gen 0:  [1000]
Gen 1:  [1100]
Gen 2:  [650, 650]
Gen 3:  [800, 800]
(4,)
>>>
```

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 3.16 where the function form of operator `splitSubPops` is used.

Example 3.16: Split subpopulations by individual information field

```
>>> import random
>>> pop = 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')
>>> SplitSubPops(pop, subPops=[0, 2], infoFields=['x'])
>>> print pop.subPopSizes()
```

```
(236, 245, 254, 265, 1000, 236, 245, 254, 265)
>>> print pop.subPopNames()
('a', 'a', 'a', 'a', 'b', 'c', 'c', 'c', 'c')
>>>
```

3.3.3 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 3.17 simulates such an evolutionary scenario. A demographic model could be added similar to Example 3.15.

Example 3.17: Merge multiple subpopulations into a single subpopulation

```
>>> simu = simulator(population([500]*2),
...     randomSelection())
>>> simu.evolve(
...     ops = [
...         mergeSubPops(subPops=[0, 1], at=3),
...         stat(popSize=True),
...         pyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     gen = 5
... )
Gen 0:  [500, 500]
Gen 1:  [500, 500]
Gen 2:  [500, 500]
Gen 3:  [1000]
Gen 4:  [1000]
(5,)
>>>
```

3.3.4 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 3.18 shows a scenario where two subpopulations expand instantly at generation 3.

Example 3.18: Resize subpopulation sizes

```
>>> simu = simulator(population([500]*2),
...     randomSelection())
>>> simu.evolve(
...     ops = [
...         resizeSubPops(proportions=(1.5, 2), at=3),
...         stat(popSize=True),
...         pyEval(r'"Gen %d:\t%s\n" % (gen, subPopSize)')
...     ],
...     gen = 5
... )
Gen 0:  [500, 500]
Gen 1:  [500, 500]
Gen 2:  [500, 500]
```

```
Gen 3:  [750, 1000]
Gen 4:  [750, 1000]
(5,)
>>>
>>>
```

3.4 Miscellaneous operators

3.4.1 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 3.4.1 demonstrates a typical usage of this operator

```
if hasSelection:
    sel = mapSelector(loci=[0], fitness=[1, 0.99, 0.98])
else:
    sel = noneOp()
#
simu.evolve(
    ops = [sel], # and other operators
)
```

3.4.2 Dump the content of a population (operator `dumper`)

Operator `dumper` and its function form `Dump` has been used extensively in this guide. They are prefect 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 outtputing 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. This operator by default only dump the present generation but you can set `ancGen` to a positive number or `-1` 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 3.19) presents a rather complicated usage of this operator.

Example 3.19: Dump the content of a population

```
>>> pop = population(size=[10, 10], loci=[20, 30], infoFields=['gen'],
...     ancGen=-1)
>>> pop.setVirtualSplitter(sexSplitter())
>>> pop1 = pop.clone()
>>> InitByFreq(pop, [0]*20 + [0.1]*10)
>>> pop.setIndInfo(1, 'gen')
>>> InitByFreq(pop1, [0]*50 + [0.1]*10)
>>> pop1.setIndInfo(2, 'gen')
>>> pop.push(pop1)
>>> Dump(pop, width=3, loci=[5, 6, 30], subPops=([0, 0], [1, 1]),
...     max=10, structure=False, ancGen=-1)
Subpopulation 0,0 (unnamed - Male), 6 individuals:
   0: MU  53 53 59 |  57 57 55 |  2
   1: MU  55 51 53 |  55 55 56 |  2
   2: MU  55 54 56 |  55 57 54 |  2
   3: MU  52 50 57 |  57 57 54 |  2
```

```
   4: MU  59 59 59 |  58 59 54 |  2
   6: MU  50 55 55 |  55 58 51 |  2
Subpopulation 1,1 (unnamed - Female), 6 individuals:
  10: FU  58 56 53 |  51 55 52 |  2
  14: FU  52 55 52 |  51 50 50 |  2
  15: FU  50 59 55 |  59 58 53 |  2
  16: FU  57 51 54 |  51 57 50 |  2

Ancestry population 1
Subpopulation 0,0 (unnamed - Male), 7 individuals:
   1: MU  22 27 22 |  23 24 25 |  1
   2: MU  20 28 28 |  28 23 20 |  1
   4: MU  23 21 28 |  28 23 25 |  1
   6: MU  26 29 29 |  25 20 23 |  1
   7: MU  29 20 21 |  26 26 23 |  1
   8: MU  26 23 28 |  24 26 27 |  1
   9: MU  22 27 23 |  25 25 28 |  1
Subpopulation 1,1 (unnamed - Female), 5 individuals:
  13: FU  27 22 28 |  23 27 24 |  1
  14: FU  28 23 24 |  21 27 28 |  1
  15: FU  28 22 25 |  27 25 22 |  1

>>>
>>>
```

3.4.3 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 3.20 to save population to files such as `snapshot_5_20.pop` during the evolution.

Example 3.20: Save snapshots of an evolving population

```
>>> simu = simulator(population(100, loci=[2]),
...     randomMating(), rep=5)
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.8])],
...     ops = [
...         savePopulation(output="!'snapshot_%d_%d.pop' % (rep, gen)",
...             step = 10),
...         ],
...     gen = 50
... )
(50, 50, 50, 50, 50)
>>>
```

3.4.4 Change ancestral depth of populations (operator `setAncestralDepth`)

Example 2.64 describes a two-stage evolutionary process where a random mating scheme is used in the first stage and another mating scheme is used in the second stage to prepare for pedigree ascertainment. The ancestral depth of

each population is changed to 1 before the second `simulator.evolve` call. This step can also be done using a `setAncestralDepth` operator, which simply set the ancestral depth of each population to a given depth (please refer to class `population` for a detailed explanation for *ancetral depth*). Example 3.21 demonstrates a tpical usage of this operator.

Example 3.21: Change ancestral depth during the evolution

```
>>> simu = simulator(population(100, infoFields=['father_idx', 'mother_idx']),
...     randomMating(), rep=5)
>>> simu.evolve(
...     preOps = [initByFreq([0.3, 0.7])],
...     ops = [
...         setAncestralDepth(2, at=-2),
...         parentsTagger(begin=-2)
...     ],
...     gen = 100
... )
(100, 100, 100, 100, 100)
>>> pop = simu.population(3)
>>> print pop.ancestralGens()
2
>>> print pop.ancestor(10, 1).info('father_idx')
43.0
>>>
```

3.4.5 Conditional operator (operator `ifElse`)

Operator `ifElse` provides a simple way to conditionally apply an operator. 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 occurance of certain events in a population. For example, Example 3.25 records the number of generations the frequency of an allele goes below 0.4 and beyong 0.6 before it gets lost or fixed in the population. Note that an else-operator can also be executed when the condition is not met.

Example 3.22: A conditional opeartor

```
>>> simu = simulator(
...     population(size=1000, loci=[1]),
...     randomMating(), rep=4)
>>> simu.evolve(
...     preOps = [
...         initByFreq([0.5, 0.5]),
...         pyExec('below40, above60 = 0, 0')
...     ],
...     ops = [
...         stat(alleleFreq=[0]),
...         ifElse('alleleFreq[0][1] < 0.4',
...             pyExec('below40 += 1')),
...         ifElse('alleleFreq[0][1] > 0.6',
...             pyExec('above60 += 1')),
...         ifElse('alleleFreq[0][1] == 0 or alleleFreq[0][1] == 1',
...             pyExec('stoppedAt = gen')),
...         terminateIf('alleleFreq[0][1] == 0 or alleleFreq[0][1] == 1')
...     ]
... )
(1708, 1496, 2617, 5240)
>>> for pop in simu.populations():
...     print 'Overall: %4d, below 40%%: %4d, above 60%%: %4d' % \
...         (pop.dvars().stoppedAt, pop.dvars().below40, pop.dvars().above60)
```

```
...
Overall: 1707, below 40%: 1301, above 60%:  234
Overall: 1495, below 40%:    0, above 60%:  935
Overall: 2616, below 40%: 1716, above 60%:   10
Overall: 5239, below 40%: 1477, above 60%: 2468
>>>
```

If more complicated logic is involved, a Python operator (`pyOperator`) should be used.

3.4.6 Turn on and off debugging mode (operator `turnOnDebug` and `turnOffDebug`)

Debug information can be useful when something looks suspicious. By turnning on certain debug code, simuPOP will print out some internal information before and during evolution. The usually way to turn on and off debug information is to use functions `TurnOnDebug(code)` and `TurnOffDebug(code)`, or setting environmental variable `SIMUDEBUG=code` where `code` is one of the debug codes listed by function `ListDebugCodes`. Note that debug information is only available in standard modules.

However, the amount of output can be overwhelming in some cases which makes it necessary to limit the debug information to certain generations. Example 3.23 demonstrates how to turn on debug information conditionally and turn it off afterwards, using operators `turnOnDebug` and `turnOffDebug`.

Example 3.23: Turn on and off debug information during evolution.

```
>>> simu = simulator(population(100, loci=[1]), randomMating(), rep=5)
>>> simu.evolve(
...     preOps = [initByFreq([0.1, 0.9])],
...     ops = [
...         stat(alleleFreq=[0]),
...         ifElse('alleleNum[0][0] == 0',
...             turnOnDebug(DBG_MUTATOR),
...             turnOffDebug(DBG_MUTATOR)),
...         ifElse('alleleNum[0][0] == 0',
...             pointMutator(loci=[0], toAllele=0, inds=[0])),
...     ],
...     gen = 100
... )
Mutate locus 0 to allele  at generation 27
Mutate locus 0 to allele  at generation 28
Mutate locus 0 to allele  at generation 32
Mutate locus 0 to allele  at generation 39
```

3.4.7 Pause and resume an evolutionary process (operator `pause`)

If you are presenting an evolutinary 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 3.24.

Example 3.24: Pause the evolution of a simulation

```
>>> simu = simulator(population(100), randomMating(), rep=10)
```

```
>>> simu.evolve(
...     preOps = [initByFreq([0.5, 0.5])],
...     ops = [pause(stopOnKeyStroke=str(x), rep=x) for x in range(10)],
...     gen = 100
... )
(100, 100, 100, 100, 100, 100, 100, 100, 100, 100)
>>>
```

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.

3.4.8 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 set its stage to `prePostMating`, 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 3.25.

Example 3.25: Monitor the performance of operators

```
>>> simu = simulator(population(10000, loci=[100]*5), randomMating(), rep=2)
>>> simu.evolve(
...     preOps = [initByFreq([0.1, 0.9])],
...     ops = [
...         stat(alleleFreq=[0]),
...         ticToc(step=50, rep=-1),
...     ],
...     gen = 101
... )
Elapsed Time: 2s  Overall Time: 00:00:02
Elapsed Time: 3s  Overall Time: 00:00:05
Elapsed Time: 4s  Overall Time: 00:00:09
(101, 101)
>>>
```

3.5 Mutation

3.5.1 k-allele mutation model

Example 3.26: A k-allele mutation model

```
>>> pop = population(size=[200, 300], ploidy=2, loci=[5, 10],
...     lociPos=[range(0, 5), range(0, 20, 2)],
...     alleleNames=['A', 'C', 'T', 'G'])
>>> simu = simulator(pop, randomMating(), rep=3)
>>> simu.evolve(
...     preOps = [initByFreq([.8, .2])],
...     ops = [
...         stat(alleleFreq=[0, 1], Fst=[1], step=10),
...         kamMutator(rate=0.001, rep=1),
...         kamMutator(rate=0.0001, rep=2)
...     ],
```

```
...     gen=10
... )
(10, 10, 10)
>>>
```

3.5.2 Stepwise mutation model

Example 3.27: A stepwise mutation model

```
>>> simu = simulator(population(size=300, loci=[3, 5]), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq( [.2, .3, .5])],
...     ops = [
...         smmMutator(rate=1,  incProb=.8),
...     ],
...     gen=1
... )
(1,)
>>>
```

3.5.3 Generalized stepwise mutation model

Example 3.28: A generalized stepwise mutation model

```
>>> import random
>>> simu = simulator(population(size=300, loci=[3, 5]), randomMating())
>>> simu.evolve(
...     preOps = [initByFreq( [.2, .3, .5])],
...     ops = [
...         gsmMutator(rate=1, p=.8, incProb=.8),
...     ],
...     gen=1
... )
(1,)
>>>
>>> def rndInt():
...   return random.randrange(3, 6)
...
>>> simu.evolve(
...     preOps = [initByFreq( [.2, .3, .5])],
...     ops = [
...         gsmMutator(rate=1, func=rndInt, incProb=.8),
...     ],
...     gen=1
... )
(1,)
>>>
```

3.5.4 Hybrid mutation model

Example 3.29: A hybrid mutation model

```
>>> import random
>>> simu = simulator(population(size=300, loci=[3, 5]), randomMating())
>>>
>>> def mutateTo(allele):
...   return allele + random.randrange(3, 6)
...
>>> simu.evolve(
```

```
...     preOps = [initByValue([50]*3 + [100]*5)],
...     ops = [
...         pyMutator(rate=0.001, func=mutateTo),
...     ],
...     gen=1
... )
(1,)
>>>
```

3.6 Selection

It is not very clear that our method agrees with the traditional 'average number of offspring' definition of fitness. (Note that this concept is very difficult to simulate because we do not know who will determine the number of offspring if two parents are involved.) We can, instead, look at the consequence of selection in a simple case (as derived in any population genetics textbook):

At generation t, genotype P_{11}, P_{12}, P_{22} has fitness values $w_{11}, w_{12,}w_{22}$ respectively. In the next generation the proportion of genotype P_{11} etc., should be

$$\frac{P_{11}w_{11}}{P_{11}w_{11} + P_{12}w_{12} + P_{22}w_{22}}$$

Now, using the 'ability-to-mate' approach, for the sexless case, the proportion of genotype 11 will be the number of 11 individuals times its probability to be chosen:

$$n_{11}\frac{w_{11}}{\sum_{n=1}^{N} w_n}$$

This is, however, exactly

$$n_{11}\frac{w_{11}}{\sum_{n=1}^{N} w_n} = n_{11}\frac{w_{11}}{n_{11}w_{11} + n_{12}w_{12} + n_{22}w_{22}} = \frac{P_{11}w_{11}}{P_{11}w_{11} + P_{12}w_{12} + P_{22}w_{22}}$$

The same argument applies to the case of arbitrary number of genotypes and random mating.

The following operators, when applied, will set a variable `fitness` and an indicator so that selector-aware mating scheme can select individuals according to these values. This has two consequences:

- Selector only set information field and mark subpopulations as selection ready. However, how these information are used to select parents can vary from mating scheme to mating scheme. As a matter of fact, some mating schemes do not support selection at all.
- selector has to be `PreMating` operator. This is not a problem when you use the operator form of the selectors since their default stage is `PreMating`. However, if you use the function form of these selectors in a `pyOperator`, make sure to set the stage of `pyOperator` to `PreMating`.

The example for `class mapSelector` is a typical example of heterozygote superiority. When $w_{11} < w_{12} > w_{22}$, the genotype frequencies will go to an equilibrium state. Theoretically, if $s_1 = w_{12} - w_{11}$ and $s_2 = w_{12} - w_{22}$, the stable allele frequency of allele 1 is

$$p = \frac{s_2}{s_1 + s_2}$$

Which is .677 in the example ($s_1 = .1$, $s_2 = .2$).

3.6.1 Map selector

Example 3.30: A selector that uses pre-defined fitness value

```
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[1], infoFields=['fitness']),
...     randomMating())
>>> s1 = .1
>>> s2 = .2
>>> simu.evolve(
...     preOps=[initByFreq(alleleFreq=[.2, .8])],
...     ops = [
...         stat(alleleFreq=[0], genoFreq=[0]),
...         mapSelector(loci=0, fitness={'0-0':(1-s1), '0-1':1, '1-1':(1-s2)}),
...         pyEval(r"'%.4f\n' % alleleFreq[0][1]", step=100)
...     ],
...     gen=300
... )
0.7665
0.3205
0.3460
(300,)
>>>
```

3.6.2 Multi-allele selector

Example 3.31: A multi-allele selector

```
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[1], infoFields=['fitness']),
...     randomMating())
>>> s1 = .1
>>> s2 = .2
>>> simu.evolve(
...     preOps=[initByFreq(alleleFreq=[.2, .8])],
...     ops = [
...         stat(alleleFreq=[0], genoFreq=[0]),
...         maSelector(loci=0, fitness=[1-s1, 1, 1-s2]),
...         pyEval(r"'%.4f\n' % alleleFreq[0][1]", step=100)
...     ],
...     gen = 300)
0.7695
0.3405
0.3645
(300,)
>>>
```

3.6.3 Multi-loci selector

Example 3.32: A multi-loci selector

```
>>> simu = simulator(
...     population(size=10, ploidy=2, loci=[2],
...     infoFields=['fitness', 'spare']),
...     randomMating())
>>> simu.evolve(
...     [ mlSelector([
...          mapSelector(loci=0, fitness={'0-0':1, '0-1':1, '1-1':.8}),
...          mapSelector(loci=1, fitness={'0-0':1, '0-1':1, '1-1':.8}),
...          ], mode = Additive),
...     ],
...     preOps = [
```

```
...         initByFreq(alleleFreq=[.2, .8])
...     ],
...     gen = 2
... )
(2,)
>>>
```

3.6.4 A hybrid selector

Example 3.33: A hybrid selector

```
>>> simu = simulator(
...     population(size=1000, ploidy=2, loci=[3], infoFields=['fitness'] ),
...     randomMating()
... )
>>>
>>> s1 = .2
>>> s2 = .3
>>> # the second parameter gen can be used for varying selection pressure
>>> def sel(arr, gen=0):
...   if arr[0] == 1 and arr[1] == 1:
...     return 1 - s1
...   elif arr[0] == 1 and arr[1] == 2:
...     return 1
...   elif arr[0] == 2 and arr[1] == 1:
...     return 1
...   else:
...     return 1 - s2
...
>>> # test func
>>> print sel([1, 1])
0.8
>>>
>>> simu.evolve(
...     preOps=[initByFreq(alleleFreq=[.2, .8])],
...     ops = [
...         stat(alleleFreq=[0], genoFreq=[0]),
...         pySelector(loci=[0, 1], func=sel),
...         pyEval(r"'%.4f\n' % alleleFreq[0][1]", step=25)
...     ],
...     gen=100
... )
0.8495
0.9855
1.0000
1.0000
(100,)
>>>
```

3.7 Recombination and Gene conversion

3.7.1 Recombination

3.7.2 Recombination with gene conversion

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
- Resulution 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.

Translated to simulation, recombination and conversion are performed in the following steps

1. Users specify the following paremters to a recombinator:
   (a) recombination points (recombinations are allowed after specified markers) (`loci`),
   (b) recombination rates (can vary from marker to marker) (`rates`),
   (c) probability of conversion if a recombination event happens (`convProb`),
   (d) track length parameters (`convMode` and `convParam`, will discuss later).
2. Starting with two parental chromosomes, randomly choose one of them to copy to an offspring chromosome until a recombination event happens.
3. This recombination event is a conversion event if
   (a) A random uniform number U(0,1) is less than the probability of conversion
   (b) The length of flanking regions does not exceed the end of chromosome

   If a conversion happens, record the end of flanking region as another recombination event.
4. Copy from another copy of parental chromosome (recombination happens), until the recorded second recombination event is reached, or another recombination event happens.
5. Repeat these steps for all chromosomes.

The tract length of a flanking region is determined by parameters `convMode` and `convParam`. `convMode` can be

- `CONVERT_NumMarkers` Convert a fixed number (`convParam`) of markers. This is the default mode with `convParam=1`.
- `CONVERT_TractLength` Convert a fixed length (`convParam`) of chromosome regions. This can be used when markers are not equally spaced on chromosomes.
- `CONVERT_GeometricDistribution` Convert a random number of markers, with a geometric distribution with parameter `convParam`.
- `CONVERT_ExponentialDistribution` Convert Convert a random length of chromosome region, using a exponential distribution with parameter `convParam`.

Note that

- If tract length is determined by length (`CONVERT_TractLength` or `CONVERT_ExponentialDistribution`), the starting point of the flanking region is uniformly distributed between marker i and $i-1$, if the recombination happens at marker i. That is to say, it is possible that no marker is converted with positive tract length.

- A conversion event will act like a recombination event if its flanking region exceeds the end of chromosome, or if another recombination event happens before the end of the flanking region.

Although any parameters can be used in a recombinator, it is worth noting that

- The probability of conversion event among all recombination events if usually expressed as ratio of conversion to recombination events in the literature. This varies greatly from study to study, ranging from 0.1 to 15 (Chen et al, Nature Review Genetics, 2007). This translate to 0.1/0.9~0.1 to 15/16~0.94 of this parameter. When \c convProb is 1, all recombination events will be conversion events. The default value if `convProb=0`, meaning no conversion.
- Conversion tract length is usually short, and is estimated to be between 337 and 456 bp, with overall range between maybe 50 - 2500 bp. simuPOP does not impose a unit for marker distance so your choice of `convParam` needs to be consistent with your unit. In the HapMap dataset, cM is usually assumed and marker distances are around 10kb (0.001cM ~- 1kb). At this marker density, gene conversion can largely be ignored.

3.8 Penetrance

3.8.1 Map penetrance model

Example 3.34: A penetrance model that uses pre-defined fitness value

```
>>> pop = population(size=[2,8], ploidy=2, loci=[2] )
>>> InitByFreq(pop, [.2, .8])
>>> MapPenetrance(pop, loci=0,
...     penetrance={'0-0':0, '0-1':1, '1-1':1})
>>> Stat(pop, numOfAffected=1)
>>>
```

3.8.2 Multi-loci penetrance model

Example 3.35: A multi-loci penetrance model

```
>>> pop = population(1000, loci=[3])
>>> InitByFreq(pop, [0.3, 0.7])
>>> pen = []
>>> for loc in (0, 1, 2):
...     pen.append(maPenetrance(loci=loc, wildtype=[1],
...         penetrance=[0, 0.3, 0.6] ) )
...
>>> # the multi-loci penetrance
>>> MlPenetrance(pop, mode=Multiplicative, peneOps=pen)
>>> Stat(pop, numOfAffected=True)
>>> print pop.dvars().numOfAffected
8
>>>
```

3.8.3 Hybrid penetrance model

Example 3.36: A hybrid penetrance model

```
>>> pop = population(1000, loci=[3])
>>> InitByFreq(pop, [0.3, 0.7])
>>> def peneFunc(geno):
...     p = 1
...     for l in range(len(geno)/2):
```

```
...         p *= (geno[1*2]+geno[1*2+1])*0.3
...     return p
...
>>> PyPenetrance(pop, func=peneFunc, loci=(0, 1, 2))
>>> Stat(pop, numOfAffected=True)
>>> print pop.dvars().numOfAffected
77
>>> #
>>> # You can also define a function, that returns a penetrance
>>> # function using given parameters
>>> def peneFunc(table):
...     def func(geno):
...       return table[geno[0]][geno[1]]
...     return func
...
>>> # then, given a table, you can do
>>> PyPenetrance(pop, loci=(0, 1, 2),
...     func=peneFunc( ((0, 0.5), (0.3, 0.8)) ) )
>>>
```

Chapter 4

Utility Modules (under revision)

4.1 simuOpt

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 2.1 and 2.2 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.

The `simuOpt` module also provides a class `simuOpt` to help users handle and manage script parameters. There are many other standard or third-party parameter handling modules in Python but this class is designed to help users run a simuPOP script in both batch and GUI modes, using a combination of parameter determination methods. More specifically, if a script uses the `simuOpt.simuOpt` class to handle parameters,

- By default, a parameter input dialog is used to accept user input if the script is executed directly. Default values are given to each parameter and users are allowed to edit them using standard parameter input widgets (on/off button, edit box, dropdown list etc). Detailed explanations to parameters are available as tooltips of corresponding input widgets. A help button is provided that will display the usage of the script when clicked.
- If a configuration file is saved for a previous simulation, command line option `--config configFile` can be used to load all parameters from that configuration file. The parameter input dialog is still used to review and modify parameters.
- Each parameter can also be set using command line options. Command line inputs will override values read from a configuration file.
- If command line option `--gui=False` is given, the script will work in batch mode. If the value of a parameter cannot be determined through command line or a configuration file, and is set not to use its default value, users will be asked to enter its value interactively. For example, `myscript.py --gui=False --config configFile` will execute a previous simulation directly.

The following sections describes how to use the `simuOpt` class in a simuPOP script.

4.1.1 Define a parameter specification list.

A `simuOpt` object is created from a list of parameter specification dictionaries, and optional short and long descriptions of a script. Each parameter specification dictionary consits of mandatory fields `longarg` (long command line

argument) and `default` (default value for this parameter) and optional fields such as `label` (label to display in the parameter input dialog and as prompt for user input), `description` (a detailed description), `allowedTypes` (allowed types of input parameter), `validate` (a function that return tells if a user input is valid), `chooseOneOf` (tells the parameter input dialog to allow users to choose one of the provided values) and `chooseFrom` (tells the parameter input dialog to allow users to choose one or more values from the provided values). Although it can be lengthy to describe a parameter in this way, it is a self-documentary process from which your users and even yourself will benefit.

Example 4.2 shows a parameter specification list that defines parameter `help`, `rate`, `rep` and `pops`. What is special about each parameter is that `help` will not be listed in the parameter input dialog (no `label`) and setting `help` to `True` during interactive parameter input will ignore all other options (`jump`); `rate` has to be between 0 and 1 (using a validation function `valueBetween`), `rep` has to be a positive integer, and `pops` can be one of the three HapMap populations. Please refer to the simuPOP reference manual for details about each dictionary key. The `description` of parameter `pop` demonstrates a special rule in the formatting of such description texts, namely **lines with symbol `'|'` as the first non-space/tab character are outputed as a separate line without the leading `'|'` character**.

Example 4.1: A sample parameter specification list

```
import types, simuOpt

options = [
    {'arg': 'r:',
    'longarg': 'rate=',
     'default': [0.01],
     'useDefault': True,
     'label': 'Recombination rate',
     'allowedTypes': [types.ListType, types.TupleType],
     'description': '''Recombination rate for each replicate. If a single value
            is given, it will be used for all replicates.''',
     'validate': simuOpt.valueListOf(simuOpt.valueBetween(0, 1))
    },
    {'longarg': 'rep=',
     'default': 5,
     'label': 'Number of replicates',
     'allowedTypes': [types.IntType, types.LongType],
     'description': 'Number of replicates to simulate.',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'pop=',
     'default': 'CEU',
     'label': 'Initial population',
     'allowedTypes': [types.StringType],
     'description': '''Use one of the HapMap populations as the initial
            population for this simulation. You can choose from:
            |YRI: 33 trios from the Yoruba people in Nigeria (Africa)
            |CEU: 30 trios from Utah with European ancestry (European)
            |CHB+JPT: 90 unrelated individuals from China and Japan (Asia)
            ''',
     'chooseOneOf': ['CEU', 'YRI', 'CHB+JPT'],
     'validate': simuOpt.valueOneOf(['CEU', 'YRI', 'CHB+JPT'])
    }
]
```

4.1.2 Get parameters

A `simuOpt` object can be created from a parameter specification list. A few member functions are immediately usable. For example, `simuOpt.usage()` returns a detailed usage message about the script and all its parameters

(although the usage message will be displayed automatically if command line option `-h` or `--help` is detected). The parameters become attributes of this object using `longarg` names so that you can access them easily (e.g. `par.rate`). Not surprosingly, all parameters now have the default value you assigned to them.

The `simuOpt.simuOpt` class provides a number of member functions that allow you to acquire user input in a number of ways. For example `simuOpt.loadConfig` reads a configuration file, `simuOpt.processArgs` checks commandline options, `simuOpt.termGetParam` asks user input interactively, and `simuOpt.guiGetParam` generates and uses a parameter input dialog. These functions can be used several times, on different sets of parameters. In addition, new options could be added programmatically using function `simuOpt.addOption` and allows further flexibility on how parameters are generated. Please refer to *the simuPOP reference manual* on how to use these functions.

Example 4.2: Get prameters using function getParam

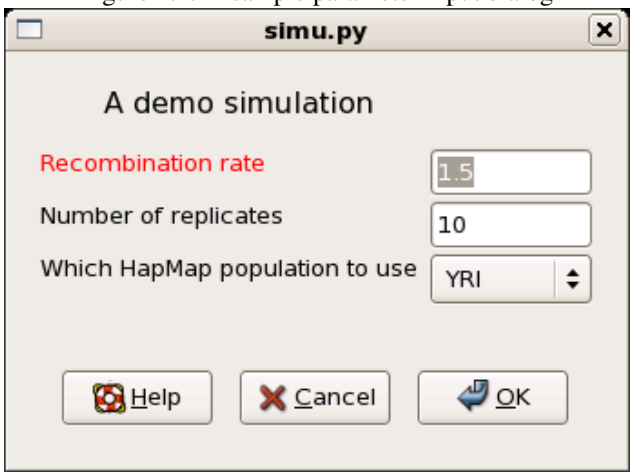
```
>>> print pars.usage()
A demo simulation

usage: runSampleCode.py [-opt [arg] | --opt [=arg]] ...

options:
  -h, --help            show this help message and exit
  --config ARG          load parameters from ARG
  --optimized           run the script using an optimized simuPOP module
  --gui ARG             which graphical toolkit to use
  -r ARG, --rate ARG    Recombination rate [default: [0.01] ]
                        Recombination rate for each replicate. If a single value
                        is given, it will be used for all replicates.
  --rep ARG             Number of replicates [default: 5 ]
                        Number of replicates to simulate.
  --pop ARG             Initial population [default: CEU ]
                        Use one of the HapMap populations as the initial
                        population for this simulation. You can choose from:
                        YRI: 33 trios from the Yoruba people in Nigeria (Africa)
                        CEU: 30 trios from Utah with European ancestry (European)
                        CHB+JPT: 90 unrelated individuals from China and Japan (Asia)

>>> # You can manually feed parameters...
>>> pars.processArgs(['--rep=10'])
True
>>> pars.rep
10
>>> # but simuOpt.getParam is the easiest to used
>>> if not pars.getParam():
...     sys.exit(1)
...
```

Example 4.2 lists some methods to determine parameter values but the last function, `simuOpt.getParam()`, will be used most of the time. This function processes each parameter in the following order:

- If a short or a long command line argument exists, use the command line argument.
- If a configuration file is specified from command line (`--config configFile`), look in this configuration file for a value.
- If `useDefault` is specified, assign a default value to this parameter.
- If `--gui=False` is specified, and the value of the parameter has not be determined, ask users interactively for a value. Otherwise, a parameter input dialog is displayed. A *Tkinter* dialog is usually used but a *wxPython* dialog will be used if *wxPython* is available (unless parameter `--gui=Tkinter` is set).

`simuOpt.getParam` returns `False` if this process fails (e.g. users click cancel in the parameter input dialog). The parameter input dialog for Example 4.2 is shown in Figure 4.1.

Figure 4.1: A sample parameter input dialog



A parameter input dialog for a script that uses the same parameter specification list as Example 4.2. The command line is `simu.py --pop=YRI`. The first parameter is in red because its input is invalid.

4.1.3 Access, manipulate and extract parameters

If `simuOpt.getParam` runs successfully, the `simuOpt` object should have valid value for each parameter. They can be retrieved as attributes (such as `par.rate`) and manipulated easily. Example 4.3 demonstrates how to extend parameter `par.rate` to have the same length as `par.rep`. Additional derived parameters can be added if needed.

When there are a large number of parameters, passing this `simuOpt` object, instead of tens of parameters, is a good way to provide clean interfaces. Alternatively, you can get a list or a dictionary of parameters using member functions `simuOpt.asList()` and `simuOpt.asDict()`. Note that derived parameters are not returned because they are not listed in the parameter specification list.

Example 4.3: Use the simuOpt object

```
>>> # save prameters to a configuration file
>>> pars.saveConfig('sample.cfg')
>>> # post-process parameters
>>> pars.rate
[0.25]
>>> pars.rep
5
>>> pars.rate = pars.rate * pars.rep
>>> # extract parameters as a dictionary or a list
>>> pars.asDict()
{'rate': [0.25, 0.25, 0.25, 0.25, 0.25], 'rep': 5, 'pop': 'CEU'}
>>> pars.asList()
[[0.25, 0.25, 0.25, 0.25, 0.25], 5, 'CEU']
>>>
```

Note: Functions in a script that makes use of the `simuOpt` class may accept a `simuOpt` object directly. If you import such a script, you can usually create a `simuOpt` object using the parameter specification list in that script. Alternatively, you can create an empty `simuOpt` object and assign values to its attributes.

4.2 simuUtil

4.3 simuRPy

Chapter 5

A real example (under revision)

In this chapter, I will show you, step by step, how to write a simuPOP script. The example is a simplified version of `scripts/simuCDCV.py` which uses a python operator to calculate and save many more statistics, and use rpy to display the dynamics of disease allele frequency.

5.1 Simulation scenario

Reich and Lander [2001] 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.

I use simuPOP to simulate this evolution process and observe the allelic spectra of both types of diseases. The results are published in Peng and Kimmel [2007], which has much more detailed discussion about the simulations, and the parameters used.

5.2 Demographic model

The initial population size is set to 10,000, as suggested in the paper. The simulation will evolve 500 generations with constant population size to reach mutation-selection equilibrium. Then, the population size will increase by around 20,000 every 10 generations and reach 1,000,000 at generation 1000. The population growth takes around 12,500 years if we assume 25 years per generation.

5.3 Mutation model

The maximum number of alleles at each locus is set to be 2000, a number that is hopefully big enough to mimic the infinite allele model. Allele 0 is the wild type (A) and all others are disease alleles (a). The $k-$allele mutation model is used. That is to say, an allele can mutate to any other allele with equal probability. An immediate implication of this model is that $P(A \to a) \gg P(a \to A)$ since there are many more a than A. The mutation rate is set to $\mu = 3.2 \times 10^{-5}$ per locus per generation.

5.4 Selection on a common and a rare disease

Two diseases are simulated: a common disease with initial allele frequency of $f_0 = 0.2$; and a rare disease with initial allele frequency of $f_0 = 0.001$. The diseases are unlinked in the sense that their corresponding loci reside on separated chromosomes. The allelic spectra of both diseases are set to be $[.9, .02, .02, .02, .02, .02]$. I.e., one allele accounts for 90% of the disease cases.

Both diseases are recessive in that their fitness values are $[1, 1, 1 - s]$ for genotype AA, Aa and aa respectively. $s_c = 0.1$, $s_r = 0.9$ are used in the simulation which imply weak selection on the common disease and strong selection on the rare disease. If an individual has both diseases, his fitness value follows a multiplicative model, i.e., $(1 - s_c) \times (1 - s_r) = 0.09$.

These parameters, translated to python, are shown in 5.1

Example 5.1: Set parameters

```
>>> initSize =  10000            # initial population size
>>> finalSize = 1000000          # final population size
>>> burnin = 500                 # evolve with constant population size
>>> endGen = 1000                # last generation
>>> mu = 3.2e-5                  # mutation rate
>>> C_f0 = 0.2                   # initial allelic frequency of *c*ommon disease
>>> R_f0 = 0.001                 # initial allelic frequency of *r*are disease
>>> max_allele = 255             # allele range 1-255 (1 for wildtype)
>>> C_s = 0.0001                 # selection on common disease
>>> R_s = 0.9                    # selection on rare disease
>>> psName = 'lin_exp'           # filename of saved figures
>>>
>>> # allele spectrum
>>> C_f = [1-C_f0] + [x*C_f0 for x in [0.9, 0.02, 0.02, 0.02, 0.02, 0.02]]
>>> R_f = [1-R_f0] + [x*R_f0 for x in [0.9, 0.02, 0.02, 0.02, 0.02, 0.02]]
>>>
```

5.5 Create a simulator

Several parameters are needed to create a population:

- `ploidy`: 2, default
- `size`: initial population size, known
- `subPop`: no subpopulation (or one single population). size can be ignored if `subPop` is given.
- `loci`: number of chromosomes and number of loci on each chromosome: we use two unlinked loci. use `loci=[1,1]`. This array gives the number of loci on each chromosome.
- loci name and position: no need to specify
- `infoFields`: This parameter is tricky since you need to specify what auxiliary information to attach to each individual. During the simulation, `fitness` is needed because all selectors generate this information and mating schemes will make use of it. If you forget to provide this parameter, never mind, the simulation will fail and tell you that a information field `fitness` is needed. Similar information fields include `father_idx` and `mother_idx` when you want to track each individual's parents using `taggers`.

You can then create a population with:

```
population(size=1000, loci=[1,1], infoFields=['fitness'])
```

To create simulator, we need to decide on a mating scheme. `randomMating` should of course be used, but we need to tell `randomMating` how population size should be changed. By default, all mating schemes keep the population size of ancestral population, but we need an instant population expansion model.

The easiest way to achieve this is defining a function that accept generation number and the population size of previous generation, and return the size of this generation. The input and output population sizes need to be arrays, indicating sizes of all subpopulations. In our case, something like `[1000]` should be used. The instant population growth model is actually quite easy to write:

```
def ins_exp(gen, oldSize=[]):
     if gen < burnin:
         return [initSize]
     else:
         return [finalSize]
```

With a little adjustment of how population size is given to `population()`, and use demographic function as a parameter to allow other demographic models to be used, we end up with example 5.2. Note that because we use loci with more than 255 allele states, the long allele module is used.

Example 5.2: Create a simulator

```
>>> from simuOpt import setOptions
>>> setOptions(alleleType='long')
>>> from simuPOP import *
>>>
>>> # instantaneous population growth
>>> def ins_exp(gen, oldSize=[]):
...     if gen < burnin:
...         return [initSize]
...     else:
...         return [finalSize]
...
>>> def simulate(incScenario):
...     simu = simulator(                                    # create a simulator
...         population(size=incScenario(0), loci=[1,1],
...             infoFields=['fitness']),                     # inital population
...         randomMating(subPopSize=incScenario)           # random mating
...     )
...
>>> #simulate(ins_exp)
>>>
```

5.6 Initialization

We start the simulation with initial allele spectra at the two loci. This can be achieved by operator `initByFreq`, which allows you to initialize individuals with alleles proportional to given allele frequencies. Using a large number of parameters, this operator can initialize any subset of loci, for any subset(s) of individuals, even given ploidy. We need only to specify locus to initialize, and use it like

```
# initialize locus 0 (for common disease)
initByFreq(atLoci=[0], alleleFreq=C_f),
# initialize locus 1 (for rare disease)
initByFreq(atLoci=[1], alleleFreq=R_f),
```

5.7 Mutation and selection

You will need to read the relative sections of the reference manual to pick suitable mutator and selectors. What we need in this case are

- k-allele mutator with given number of allele states (k). This is exactly

  ```
  kamMutator(rate=mu, maxAllele=max_allele)
  ```

- single locus selector that treat 0 as wildtype, and any other allele as mutant. The selector to use is

  ```
  maSelector(locus=0, fitness=[1,1,1-C_s], wildtype=[0])
  ```

  and

  ```
  maSelector(locus=1, fitness=[1,1,1-R_s], wildtype=[0])
  ```

- Because an individual has only one fitness value, fitness values obtained from two selectors need to be combined (another choice is that you can use a selector that handle multiple loci.). Therefore, we use a multi-locus selector as follows:

  ```
  mlSelector([
      maSelector(locus=0, fitness=[1,1,1-C_s], wildtype=[0]),
      maSelector(locus=1, fitness=[1,1,1-R_s], wildtype=[0])
      ], mode=SEL_Multiplicative)
  ```

With these operators, the simulator can be started. It first initialize a population with given allelic spectra, and then evolve it, subject to mutation and selection, specific to each locus. The program is listed in example 5.3:

Example 5.3: Run the simulator

```
>>> def simulate(incScenario):
...     simu = simulator(                       # create a simulator
...         population(size=incScenario(0), loci=[1,1],
...             infoFields=['fitness']),        # inital population
...         randomMating(subPopSize=incScenario)
...     )
...     simu.evolve(                            # start evolution
...         preOps=[                            # operators that will be applied before evolution
...             # initialize locus 0 (for common disease)
...             initByFreq(loci=[0], alleleFreq=C_f),
...             # initialize locus 1 (for rare disease)
...             initByFreq(loci=[1], alleleFreq=R_f),
...         ],
...         ops=[                               # operators that will be applied at each gen
...             # mutate: k-alleles mutation model
...             kamMutator(rate=mu, maxAllele=max_allele),
...             # selection on common and rare disease,
...             mlSelector([                # multiple loci - multiplicative model
...                 maSelector(loci=0, fitness=[1,1,1-C_s], wildtype=[0]),
...                 maSelector(loci=1, fitness=[1,1,1-R_s], wildtype=[0])
...             ], mode=SEL_Multiplicative),
...         ],
...         gen = endGen
...     )
...
>>> #simulate(ins_exp)
>>>
```

5.8 Output statistics

We first want to output total disease allele frequency of each locus. This is easy since `stat()` operator can calculate allele frequency for us. What we need to do is use `stat()` operator to calculate allele frequency and set variable `alleleFreq` (and `alleleNum`) in each population's local namespace,

```
stat(alleleFreq=[0,1]),
```

and then use a `pyEval` (python expression) operator to print out the values:

```
pyEval(r' %.3f\t%.3f\n % (1-alleleFreq[0][0], 1-alleleFreq[1][0])')
```

The `pyEval` operator can accept any valid python expression so the above expression calculate $f_0 = \sum_{i=1}^{\infty} f_i$ at each locus (0 and 1) and print it in the format of `'%.3f\t%.3f\n'`.

There is no operator to calculate effective number of alleles [Reich and Lander, 2001] so we need to do that by ourselves, using allele frequencies. The formula to calculate effective number of alleles is

$$n_e = \left(\sum_i \left(\frac{f_i}{f_0} \right)^2 \right)^{-1}$$

where f_i is the allele frequency of disease allele i , and f_0 is defined as above. To calculate n_e at the first locus, we can use a `pyEval` operator (a direct translation of the formula):

```
pyEval('1./sum([(x/(1-alleleFreq[0][0]))**2 for x in alleleFreq[0][1:]])')
```

However, this expression looks complicated and can not handle the case when $f_0 = 0$. A more complicated, and robust method is using the `stmts` parameter of `pyEval`, which will be evaluated before parameter `expr` ,

```
pyEval(stmts='''ne = [0,0]
for i in range(2):
    freq = alleleFreq[i][1:]
    f0 = 1 - alleleFreq[i][0]
    if f0 == 0:
        ne[i] = 0
    else:
        ne[i] = 1./sum([(x/f0)**2 for x in freq])
''', expr=r'%.4f\t%.4f\n % (ne[0], ne[1])')
```

As you can see, the `pyEval` can be really complicated and calculate any statistics. However, if you plan to calculate more statistics, a pure python operator may be easier to write. The simplest form of a python operator is just a python function that accept a population as the first parameter (and an optional parameter),

```
def ne(pop):
    ' calculate effective number of alleles '
    Stat(pop, alleleFreq=[0,1])
    f0 = [0, 0]
    ne = [0, 0]
    for i in range(2):
        freq = pop.dvars().alleleFreq[i][1:]
        f0[i] = 1 - pop.dvars().alleleFreq[i][0]
        if f0[i] == 0:
            ne[i] = 0
        else:
            ne[i] = 1. / sum([(x/f0[i])**2 for x in freq])
    print '%d\t%.3f\t%.3f\t%.3f\t%.3f\n' % (pop.gen(), f0[0], f0[1], ne[0], ne[1])
    return True
```

Then, you can use this function in a python operator

```
pyOperator(func=ne, step=5)
```

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 vars explicitly using the `pop.dvars()` function. (This also implies that you can do whatever you want to the population.). In this example, the function form of the `stat` operator is used to explicitly calculate allele frequency. The results are also explicitly printed using the `print` command. The explicities lead to longer, but clearer program. This becomes obvious when you need to calculate and print many statistics.

The following program (listing 5.4) uses the `pyOperator` solution. In this program, user can input two demographic models as command line parameter. Two other operators are used

- A `ticToc` operator that prints out elapsed time at every 100 generations
- A `pause` operator that pause the simulation whenever you press a key. You can actually enter a python command shell to examine the results.

Example 5.4: The whole program

```
>>> def ne(pop):
...     ' calculate effective number of alleles '
...     Stat(pop, alleleFreq=[0,1])
...     f0 = [0, 0]
...     ne = [0, 0]
...     for i in range(2):
...         freq = pop.dvars().alleleFreq[i][1:]
...         f0[i] = 1 - pop.dvars().alleleFreq[i][0]
...         if f0[i] == 0:
...             ne[i] = 0
...         else:
...             ne[i] = 1. / sum([(x/f0[i])**2 for x in freq])
...     print '%d\t%.3f\t%.3f\t%.3f\t%.3f' % (pop.gen(), f0[0], f0[1], ne[0], ne[1])
...     return True
...
>>>
```

5.9 Option handling

Everything seems to be perfect until you need to run more simulations with different parameters like initial population size. Editing the script again and again is out of the question. Since this script is a python script, it is tempting to use python modules like `getopt` to parse options from command line. A better choice would be using the `simuOpt` module. Using this module properly, your simuPOP should be able to get options from short or long command line option, from a configuration file, from a tkInter of wxPython dialog, or from user input. Taking `c:\python\share\simuPOP\scripts\simuLDDecay.py` as an example, you can run this script as follows:

- use command '`simuLDDecay.py`' or double click the program
- click the help button on the dialog, or run

  ```
  > simuLDDecay.py -h
  ```

  to view help information.

enter parameters in a parameter dialog, or use short or long command arguments

```
> simuLDDecay.py -s 500 -e 10 --recRate 0.1 --numRep 5 --gui=False
```

- use the optimized module by

  ```
  > simuLDDecay.py --optimized
  ```

save the parameters to a config file

```
> simuLDDecay.py --quiet -s 500 -e 10 --saveConfig decay.cfg
```

this will result in a config file `decay.cfg` with these parameters.

- and of course use `-c` or `--config`,

  ```
  > simuLDDecay.py --config decay.cfg
  ```

  to load parameters from the config file.

The last function is very useful since you frequently need to run many slightly different simulations, saving a configuration file along with your results will make your life much easier.

To achieve all the above, you need to write your scripts in the following order:

1. First line:

   ```
   #!/usr/bin/env python
   ```

2. Write the introduction of the whole script in a module-wise doc string.

   ```
   '''
   This script will ....
   '''
   ```

   These comments can be accessed as module `__doc__` and will be displayed as help message.

3. Define an option data structure.

   ```
   options = [
   ... a dictionary of all user input parameters ...
   ]
   ```

   These parameters will be handled by simuPOP automatically. Users will be able to set them through command line, configuration file, Tkinter- or wxPython-based GUI. The detailed description of this structure is given in simuPOP reference manual.

4. Main simulation functions

5. In the executable part of the script (under `__name__ == '__main__'`), you should call `simuOpt.getParam` to let `simuOpt` handle all parameter input for you and obtain a list of parameters. You usually need to handle some special cases (`-h`, `--saveConfig` etc), and they are all standard.

You will notice that `simuOpt` does all the housekeeping things for you, including parameter reading, conversion, validation, print usage, save configuration file. Since most of the parts are pretty standard, you can actually copy any of the scripts under the `scripts` directory as a template for your new script. The following example 5.5 shows the beginning and the execution part of the complete `reich.py` script, which can be found under the doc directory. For a complete reference of the Options structure, please refer to the reference manual.

Example 5.5: Option handling

```
>>> options = [
...     {'arg': 'h',
...      'longarg': 'help',
...      'default': False,
...      'description': 'Print this usage message.',
...      'allowedTypes': [types.NoneType, type(True)],
...      'jump': -1                    # if -h is specified, ignore any other parameters.
...     },
...     {'longarg': 'initSize=',
...      'default': 10000,
...      'label': 'Initial population size',
...      'allowedTypes': [types.IntType, types.LongType],
...      'description': '''Initial population size. This size will be maintained
...                till the end of burnin stage''',
...      'validate': simuOpt.valueGT(0)
...     },
...     {'longarg': 'finalSize=',
...      'default': 1000000,
...      'label': 'Final population size',
...      'allowedTypes': [types.IntType, types.LongType],
...      'description': 'Ending population size (after expansion.',
...      'validate': simuOpt.valueGT(0)
...     },
...     {'longarg': 'burnin=',
...      'default': 500,
...      'label': 'Length of burn-in stage',
...      'allowedTypes': [types.IntType],
...      'description': 'Number of generations of the burn in stage.',
...      'validate': simuOpt.valueGT(0)
...     },
...     {'longarg': 'endGen=',
...      'default': 1000,
...      'label': 'Last generation',
...      'allowedTypes': [types.IntType],
...      'description': 'Ending generation, should be greater than burnin.',
...      'validate': simuOpt.valueGT(0)
...     },
...     {'longarg': 'growth=',
...      'default': 'instant',
...      'label': 'Population growth model',
...      'description': '''How population is grown from initSize to finalSize.
...                Choose between instant, linear and exponential''',
...      'chooseOneOf': ['linear', 'instant'],
...     },
...     {'longarg': 'name=',
...      'default': 'cdcv',
...      'allowedTypes': [types.StringType],
...      'label': 'Name of the simulation',
...      'description': 'Base name for configuration (.cfg) log file (.log) and figures (.eps)'
...     },
... ]
>>>
>>> def getOptions(details=__doc__):
...     # get all parameters, __doc__ is used for help info
...     allParam = simuOpt.getParam(options,
...         'This program simulates the evolution of a common and a rare direse\n' +
...         'and observe the evolution of allelic spectra\n', details)
...     #
```

```
...     # when user click cancel ...
...     if len(allParam) == 0:
...         sys.exit(1)
...     # -h or --help
...     if allParam[0]:
...         print simuOpt.usage(options, __doc__)
...         sys.exit(0)
...     # automatically save configurations
...     name = allParam[-1]
...     if not os.path.isdir(name):
...         os.makedirs(name)
...     simuOpt.saveConfig(options, os.path.join(name, name+'.cfg'), allParam)
...     # return the rest of the parameters
...     return allParam[1:-1]
...
>>> #
>>> # IGNORED
>>> #
>>>
>>> if __name__ == '__main__':
...     # get parameters
...     (initSize, finalSize, burnin, endGen, growth) = getOptions()
...     #
...     from simuPOP import *
...     #
...     if initSize > finalSize:
...         print 'Initial size should be greater than final size'
...         sys.exit(1)
...     if burnin > endGen:
...         print 'Burnin gen should be less than ending gen'
...         sys.exit(1)
...     if growth == 'linear':
...         simulate(lin_exp)
...     else:
...         simulate(ins_exp)
...
>>>
```

Bibliography

Bo Peng and Marek Kimmel. Simulations provide support for the common disease common variant hypothesis. *Genetics*, 175:1–14, 2007. 95

Bo Peng, Chris I Amos, and Marek Kimmel. Forward-time simulations of human populations with complex diseases. *PLoS Genetics*, 3:e47, 2007. 50

David E Reich and Eric S Lander. On the allelic spectrum of human disease. *Trends Genet*, 17(9):502–510, 2001. 95, 99

Neil Risch. Linkage strategies for genetically complex traits. i. multilocus models. *Am J Hum Genet*, 46:222–228, 1990. 32

Index