simuPOP User's Guide

Release 0.9.5 (Rev: 2834)

Bo Peng

December 2004

Last modified
July 31, 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

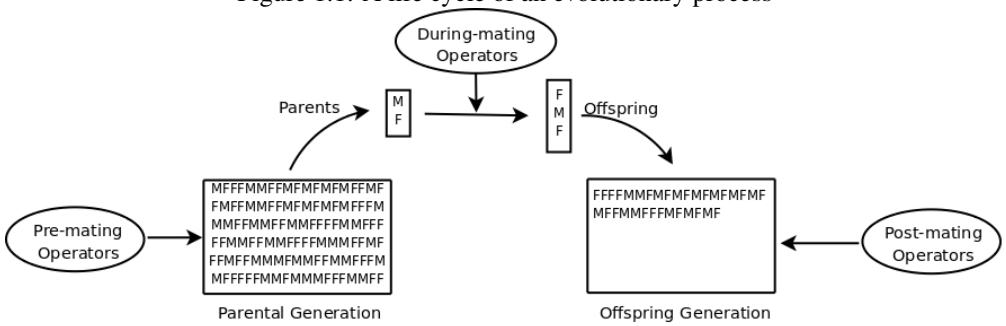


Illustration of the discrete-generation evolutionary model used by simuPOP.

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(rates=0.01),
...         stat(LD=[0, 1]),
...         pyEval(r"'%.2f\t' % LD[0][1]", step=10),
...         pyOutput('\n', reps=-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.addInfoFields)
Help on method population_addInfoFields in module _simuPOP_std:

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

        x.addInfoFields(fields, init=0)

    Details:

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

```
>>>
```

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. 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. Because some sections describe advanced features that are only used in the construction of highly complex simulations, or implementation details that concern only advanced users, new simuPOP users can safely skip these sections. **Sections that describe advanced topics are marked by one or two asterisks (*) after the section heads**.

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), `reps` (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.

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 *
>>> 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:524 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:468 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. **Note:** Please do not make use of exceptions raised by simuPOP functions to direct the logic of your script (e.g. use a `try ... except ...` statement around function `infoIdx` to find a valid information field). Because the optimized modules do not raise these exceptions, such a script may crash or yield invalid results when the optimized module is 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. `GetRNG().seed()`) and use it to replay the session later. **Note:** `GetRNG().seed()` returns the seed of the simuPOP random number generator. It can be used to replay your simulation if `GetRNG()` is your only source of random number generator. If you also use the Python `random` module, it is a good practise to set its seed using `random.seed(GetRNG().seed())`.

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 2.15 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 2.15 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, 0]
>>> print geno
[1, 1, 1, 0, 2, 0, 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, 0]
>>> print geno              # but not geno
[1, 1, 1, 0, 2, 0, 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, 0]
>>>
```

2.2.5 `defdict` datatype

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

To address this problem, a special default dictionary type `defdict` is used for dictionaries with keys determined from a population. This derived dictionary type works just like a regular dictionay, but it returns 0, instead of raising a `KeyError` exception, when an invalid key is used. For example, subpopulations in Example 2.7 have different alleles. Although `pop.dvars(sp).alleleFreq[0]` have only two keys for `sp=0` or `1`, `pop.dvars(sp).alleleFreq[0][x]` are used to print frequencies of alleles `0`, `1` and `2`.

Example 2.7: The defdict datatype

```
>>> pop = population([100]*2, loci=1)
>>> InitByFreq(pop, [0, 0.2, 0.8], subPops=0)
>>> InitByFreq(pop, [0.2, 0.8], subPops=1)
>>> Stat(pop, alleleFreq=0, vars=['alleleFreq_sp'])
>>> for sp in range(2):
...     print 'Subpop %d (with %d alleles): ' % (sp, len(pop.dvars(sp).alleleFreq[0])),
...     for a in range(3):
```

```
...         print '%.2f ' % pop.dvars(sp).alleleFreq[0][a],
...     print
...
Subpop 0 (with 2 alleles):  0.00  0.16  0.84
Subpop 1 (with 2 alleles):  0.20  0.80  0.00
>>>
```

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

2.2.6 Parameter names and single and list input of parameters

simuPOP follows (at least tries to follow) the following naming convention:

- If a parameter only accept a single input, singular names such as `field`, `locus`, `value`, and `name` are used.
- If a parameter accepts a list of values, plural names such as `fields`, `loci`, `values` and `names` are used.
- **Plural form parameters usually accept single inputs.** For example, `loci=1` can be used as a shortcut for `loci=[1]` and `infoFields='x'` can be used as a shortcut for `infoFields=['x']`.

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

2.3 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()`).

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

- A codon can be simulated as a locus with 64 allelic states, or three locus each with 4 allelic states. Alleles in the first case would be codons such as `AAC` and a mutation event would mutate one codon to another (e.g. `AAC` -> `ACC`). Alleles in the second case would be `A`, `C`, `T` or `G`, and a mutation event would mutate one nucleotide to another (e.g. `A` -> `G`).
- You can use 0 and 1 (and the binary module of simuPOP) to simulate SNP (single-nucleotide polymorphism) markers and ignore the exact meaning of 0 and 1, or use 0, 1, 2, 3 to simulate different nucleotide (A, C, T, or G) in these markers. The mutation model in the second case would be more complex.
- For microsatellite markers, alleles are usually interpreted as the number of tandem repeats. It would be difficult (though doable) to simulate the expansion and contraction of genome caused by the mutation of microsatellite markers.

- The infinite site and infinite allele mutation models could be simulated using either a continuous sequence of nucleotides with a simple 2-allele mutation model, or a locus with a large number of possible allelic states. It is also possible to simulate an empty region (without any locus) with loci introduced by mutation events.
- If you consider deletion as a special allelic state, you can simulate gene deletions without shrinking a chromosome. For example, a deletion mutation event can set the allelic state of one or more loci to 0, which can no longer be mutated.

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

A genotypic structure can be retrieved from *individual* and *population* objects. Because a population consists of individuals of the same type, genotypic information can only be changed for all individuals at the population level. Populations in a simulator usually have the same genotypic structure because they are created by as replicates, but their structure may change during evolution. Example 2.15 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 2.8: 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)
''
>>> # 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')
>>>
```

Note: simuPOP does not assume any unit for loci positions. Depending on your application, it can be basepair (bp), kilo-basepair (kb), mega base pair (mb) or even using genetic-map distance such as centiMorgan. It is your

responsibility to interpret and use loci positions properly. For example, recombination rate between two adjacent markers can be specified as the product between their physical distance and a recombination intensity. The scale of this intensity will vary by the unit assumed.

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

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 populations 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 2.15.

Example 2.9: 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)
   (1),  (2),  (3)
2: chrom2 (Autosome, 5 loci)
   (1),  (2),  (3),  (4),  (5)
population size: 7 (2 subpopulations with 2, 5 individuals)
Number of ancestral populations: 0

Subpopulation 0 (), 2 individuals:
   0: MU 110 11011 | ___ _____
   1: MU 100 11111 | ___ _____
Subpopulation 1 (), 5 individuals:
   2: MU 101 11010 | ___ _____
   3: FU 011 11101 | 010 11110
   4: MU 111 11010 | ___ _____
   5: FU 111 11111 | 111 00010
   6: FU 100 01101 | 010 11110
>>>
```

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

Example 2.10: 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 (), 6 individuals:
   0: FU 101 101 110000 ____ 0110 1101 | 111 100 001110 ____ 0111 1111
   1: FU 010 111 010010 ____ 0110 0011 | 011 111 111100 ____ 1001 0111
   2: FU 011 111 011011 ____ 1100 1011 | 110 101 101111 ____ 1011 1110
   3: FU 011 111 011110 ____ 1111 1001 | 111 011 100111 ____ 1111 1111
   4: FU 111 110 000100 ____ 1111 1011 | 111 101 010001 ____ 0110 0111
   5: MU 111 111 111010 ____ 1110 0011 | 001 101 _______ 0110 1111 0110
>>>
```

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.

2.3.3 Information fields

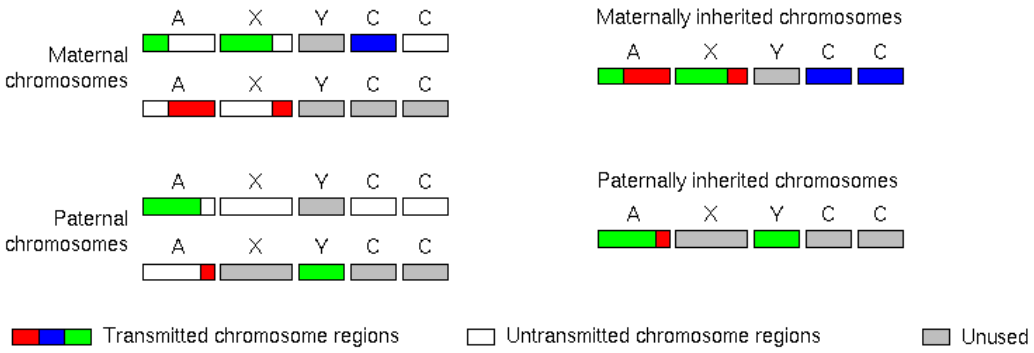
Different kinds of simulations require different kinds of individuals. Individuals with only genotype information are sufficient to simulate the basic Wright-Fisher model. Sex is needed to simulate such a model in diploid populations with sex. Individual fitness may be needed if selection is induced, 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 2.11: 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),
```

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.

```
...     ops = [
...         parentsTagger(),
...         recombinator(rates=0.01)
...     ],
...     gen = 1
... )
(1,)
>>> pop = simu.extract(0)
>>> pop.indInfo('mother_idx')  # mother of all offspring
(5.0, 1.0, 4.0, 8.0, 4.0, 5.0, 9.0, 3.0, 4.0, 6.0)
>>> ind = pop.individual(0)
>>> mom = pop.ancestor(ind.intInfo('mother_idx'), 1)
>>> print ind.genotype(0)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> print mom.genotype(0)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> print mom.genotype(1)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>>>
```

Example 2.15 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.

Information fields can be located both by names and by indexes, the former provides better readability at a slight cost of performance because these names have to be translated into indexes each time. Because `ind.info('x')` is essentially `ind.info(ind.infoIdx('x'))`, it might be a good idea to look up the index of an information field (e.g. `field = pop.infoIdx('x')`) and use this index to access the information field (e.g. `ind.info(field)`) if functions such as `ind.info('x')` will be called repeatedly in a loop.

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

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 2.13 demonstrates how to access and modify individual sex, affection status and information fields. Note that although all information fields are stored as float values, a function `intInfo` is provided to return an integer value (the same as `int(info(field))`).

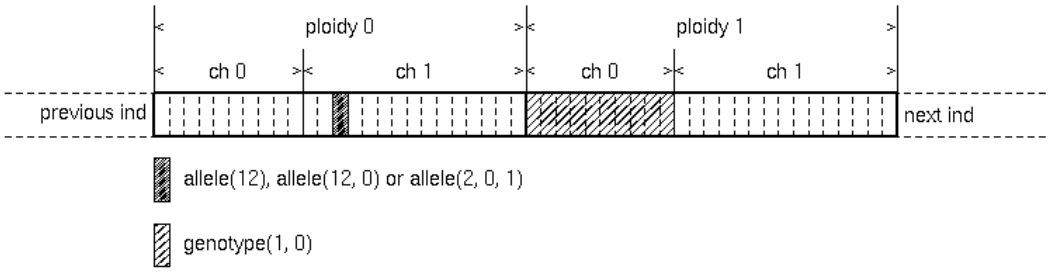
Example 2.12: Access Individual properties

```
>>> 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
>>> ind.intInfo(0)          # obtain the value of 'x' as an integer.
4
>>>
```

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.

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 2.13 demonstrates the use of these functions.

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)

Example 2.13: 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. **If a population is not structured, it has one subpopulation that is the population itself.** Subpopulations serve as barriers of individuals in the sense that mating only happens between individuals in the same subpopulation. A number of functions are provided to merge, remove, resize subpopulations, and move individuals between subpopulations (migration). You will rarely get a chance to use them directly because such operations are usually handled by operators.

Example 2.15 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 2.14: 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])
1
>>> # split subpopulations
>>> pop.splitSubPop(1, [2, 7])
(1, 2)
>>> pop.subPopSizes()
(3, 2, 7)
>>> # 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 (), 3 individuals:
   0: MU  0 |  0
   1: MU  4 |  0
   2: MU  9 |  0
Subpopulation 1 (), 3 individuals:
   3: MU  1 |  1
   4: MU  5 |  1
   5: MU 10 |  1
Subpopulation 2 (), 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 2.15: 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.splitSubPop(1, [2, 3])
(1, 2)
>>> pop.subPopNames()
('x', 'z', 'z')
>>> pop.setSubPopName('z-1', 1)
>>> pop.subPopNames()
('x', 'z-1', 'z')
>>> pop.subPopByName('z')
2
>>>
```

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 2.16 demonstrates how to apply virtual splitters to a population, and how to check VSP names and sizes.

Example 2.16: 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
'x = 0'
>>> pop.subPopSize([0, 0])     # Size of VSP 0 in subpopulation 0
58
```

```
>>> pop.subPopSize([1, 0])     # Size of VSP 0 in subpopulation 1
90
>>> # 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
84
>>>
```

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

Example 2.17: 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([Male], 'Sex', [0, 0])
>>> pop.setIndInfo([Female], 'Sex', [0, 1])
>>> # Print individual genotypes, followed by values at information field Sex
>>> Dump(pop, structure=False)
Subpopulation 0 (), 10 individuals:
   0: MU 00 000 | 00 000 |  1
   1: FU 11 111 | 11 111 |  2
   2: MU 00 000 | 00 000 |  1
   3: FU 11 111 | 11 111 |  2
   4: MU 00 000 | 00 000 |  1
   5: MU 00 000 | 00 000 |  1
   6: MU 00 000 | 00 000 |  1
   7: FU 11 111 | 11 111 |  2
   8: FU 11 111 | 11 111 |  2
   9: FU 11 111 | 11 111 |  2
>>>
```

Note: Current implementation of virtual subpopulation iterators does not allow nested use of VSP-related functions. For example:

```
for boy in pop.individuals([0, 0]):
    for girl in pop.individuals([0, 1]):
        ...
```

will yield unexpected results because `pop.individuals([0, 1])` will override individuals selected for `pop.individuals([0, 0])`.

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 than one generations are stored in a population, function `ancestor(idx, [subPop], gen)` can be used to access individual from an ancestral generation (see Section 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 2.19 demonstrates how to use all these individual-access functions.

Example 2.18: 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 2.19 demonstrates the use of such functions.

Example 2.19: 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')
(0.0, 10.0, 7.0, 6.0, 9.0, 2.0, 10.0, 1.0, 4.0, 6.0)
>>> 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)
(9.0, 2.0, 10.0, 1.0, 4.0, 6.0)
>>> pop.genotype(1)
[5, 6, 7, 5, 0, 0, 0, 0, 6, 7, 5, 6, 0, 0, 0, 0, 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`. Example 2.20 demonstrates how to read and write information fields from an individual, or from a population in batch mode. Note that functions `population.indInfo` and `population.setIndInfo` can be applied to (virtual) subpopulation using a optional parameter subPop.

Example 2.20: Add and use of information fields in a population

```
>>> pop = population(10)
>>> pop.setInfoFields(['a', 'b'])
>>> pop.addInfoFields('c')
>>> pop.addInfoFields(['d', 'e'])
>>> pop.infoFields()
('a', 'b', 'c', 'd', 'e')
>>> #
>>> cIdx = pop.infoIdx('c')
>>> eIdx = pop.infoIdx('e')
>>> # information fields can be accessed in batch mode
>>> pop.setIndInfo([1], cIdx)
>>> # as well as individually.
>>> 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 2.23. 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 2.21: 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
... )
0.406
0.410
0.409
(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   0.406
1   0.410
0   0.409
>>> # 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 2.23 demonstrates how to use these populations.

Example 2.22: 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)
   (1),  (2),  (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 (), 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. This function works as follows:

- If all default parameters are used, this function is equivalent to `population::clone()`.
- If a list of loci are given to parameter `loci`, only specified loci will be copied to the extracted population. Loci in parameter `loci` do not have to be in order but loci in the extracted population will keep their original positions, and thus in their original order.
- If a list of information fields are given to parameter `info`, only specified information fields will be copied to the extracted population.
- If a positive `ancGen` is given, only generations `0 - ancGen` will be extracted.
- Subset of individuals is specified differently. Instead of a list of individuals that will be extracted, an information field is expected. This information field should 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, parameter `ped`) with the same number of individuals is given, the information field from that population is used.

Example 2.23 demonstrates the use of this powerful function.

Example 2.23: 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 (), 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 (), 3 individuals:
   3: MU 123 67 | 123 67 |  1
   4: FU 123 67 | 123 67 |  1
   5: FU 123 67 | 123 67 |  1
Subpopulation 2 (), 4 individuals:
   6: MU 123 67 | 123 67 |  2
   7: MU 123 67 | 123 67 |  2
   8: MU 123 67 | 123 67 |  2
   9: FU 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 2.24 demonstrates how to set and access Population variables.

Example 2.24: 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: {0: 0.33500000000000002, 1: 0.66500000000000004}},
 'alleleNum': {0: {0: 67.0, 1: 133.0}},
 'myVar': 21,
 'popSize': 100,
 'subPopSize': [100]}
>>> # print number of allele 1 at locus 0
>>> print pop.vars()['alleleNum'][0][1]
133.0
>>> # use the dvars() function to access dictionary keys as attributes
>>> print pop.dvars().alleleNum[0][1]
133.0
>>> print pop.dvars().alleleFreq[0]
defdict({0: 0.33500000000000002, 1: 0.66500000000000004})
>>>
```

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 2.32 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 2.25: 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('len(alleleFreq[0]) == 1')
...       ]
... )
(154, 196, 479, 436, 143)
>>>
```

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 2.32 demonstrates how to save and load a population in the native simuPOP format.

Example 2.26: 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 of 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 2.32 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 2.27: 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: 0.00
Around gen 90: allele Freq: 0.00
After gen 90: allele freq: 0.00
Around gen 90: allele Freq: 0.00
Around gen 99: allele Freq: 0.00
Around gen 99: allele Freq: 0.00
(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 `reps` is used to control which replicate(s) an operator can be applied to. This parameter can be a list of replicate numbers or a single replicate number. Negative index is allowed where `-1` refers to the last replicate. This technique has been widely used to produce table-like output where a `pyOutput` outputs a newline when it is applied to the last replicate of a simulator. Example 2.32 demonstrates how to use this `reps` parameter. It is worth noting that negative indexes are *dynamic* indexes relative to number of active populations. For example, `rep=-1` will refer to a previous population if the last population has stopped evolving. Use a non-negative replicate number if this is not intended.

Example 2.28: 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, reps=0),
...         pyEval(r"'\t%.2f' % alleleFreq[0][0]", step=10, reps=(0, 2, -1)),
...         pyOutput('\n', step=10, reps=-1)
...     ],
...     gen=30,
... )
0       0.25    0.14    0.21
10      0.20    0.33    0.20
20      0.09    0.14    0.18
(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. Generally speaking,

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

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

2.6.3 Operator output *

All operators we have seen, except for the `savePopulation` operator in Example 2.27, 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`, ...
- A Python function that can accept a string as its only parameter (`func(msg)`). When an operator outputs a message, this function will be called with this message.

Because a table output such as the one in Example 2.32 is written by several operators, it is clear that all of them need to use the `'>>'` output format.

The `savePopulation` operator in Example 2.27 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 2.32 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 2.29: Use the output and outputExpr parameters

```
>>> simu = simulator(population(size=1000, loci=2), randomMating(), rep=3)
>>> simu.evolve(
...     preOps = initByValue([1, 2, 2, 1]),
...     ops = [
...         recombinator(rates=0.01),
...         stat(LD=[0, 1]),
...         pyEval(r"'%.2f\t' % LD[0][1]", step=20, output='>>LD.txt'),
...         pyOutput('\n', reps=-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    0.25
0.21    0.19    0.19
0.16    0.15    0.15
0.14    0.07    0.13
0.12    0.10    0.10

>>> print open('R2.txt').read()    # Only the last write operation succeed.
0.16
>>> print open('LD_2.txt').read()  # Each replicate writes to a different file.
0.25    0.19    0.15    0.13    0.10
>>>
```

Example 2.30 demonstrates an advanced usage of the `output` parameter. In this example, a logging object is created to write to a logfile as well as the standard output. The `info` and `debug` functions of this object are assigned to two operators so that their outputs can be sent to both a logfile and to the console window. One of the advantages of using a logging mechanism is that debugging output could be suppressed easily by adjusting the logging level of the logging object. Note that function `logging.info()` automatically adds a new line to its input messages before it writes them to an output.

Example 2.30: Output to a Python function

```
>>> import logging
>>> # logging to a file simulation.log, with detailed debug information
>>> logging.basicConfig(
...     filename='simulation.log',
...     level=logging.DEBUG,
...     format='%(levelname)s: %(message)s',
...     filemode='w'
... )
>>> # logging to standard output with less information
>>> console = logging.StreamHandler()
>>> console.setLevel(logging.INFO)
>>> formatter = logging.Formatter('%(message)s')
>>> console.setFormatter(formatter)
>>> logger = logging.getLogger('')
>>> logger.addHandler(console)
>>> #
>>> simu = simulator(population(size=1000, loci=2), randomMating())
>>> simu.evolve(
...     preOps = initByValue([1, 2, 2, 1]),
...     ops = [
...         recombinator(rates=0.01),
...         stat(LD=[0, 1]),
...         pyEval(r"'LD: %d, %.2f' % (gen, LD[0][1])", step=20,
...             output=logger.info),   # send LD to console and a logfile
...         pyEval(r"'R2: %d, %.2f' % (gen, R2[0][1])", step=20,
...             output=logger.debug),  # send R2 only to a logfile
...     ],
...     gen=100
... )
LD: 0, 0.25
LD: 20, 0.20
LD: 40, 0.18
LD: 60, 0.11
LD: 80, 0.08
(100,)
>>> print open('simulation.log').read()
INFO: LD: 0, 0.25
DEBUG: R2: 0, 0.96
INFO: LD: 20, 0.20
DEBUG: R2: 20, 0.64
INFO: LD: 40, 0.18
DEBUG: R2: 40, 0.50
INFO: LD: 60, 0.11
DEBUG: R2: 60, 0.19
INFO: LD: 80, 0.08
DEBUG: R2: 80, 0.11

>>>
```

2.6.4 During-mating operators and genotype transmitters

All operators in Examples 2.27, 2.28 and 2.29 are applied before or after mating. There is, however, a hidden during-mating operator that is called by `randomMating()`. This operator is called `mendelianGenoTransmitter()` and is responsible for transmitting genotype from parents to offspring according to Mendel's laws. All pre-defined mating schemes (see Section 2.7) use a special kind of during-mating operator to transmit genotypes. They are called **genotype transmitters**.

Additional during-mating operators could be applied to change or reject offspring. A good example is operator `parentsTagger()` that record the indexes of an offspring's parents in the parental generation. These operators usually do not transmit genotypes and are called after the genotype transmitter in a mating scheme set genotype for the offspring.

Only one genotype transmitter is applied if two or more transmitters are provided. If a genotype transmitter is passed from a simulator (using parameter `ops`), the genotype transmitter in the mating scheme will not be called. For example, a `recombinator` is used in Example 2.31 to transmit parental genotypes to offspring after generation 30. The default genotype transmitter `mendelianGenoTransmitter` is used for the first 30 generations.

Example 2.31: Genotype transmitters

```
>>> simu = simulator(population(size=10000, loci=2), randomMating())
>>> simu.evolve(
...     preOps = initByValue([1, 2, 2, 1]),
...     ops = [
...         # Recombination only happens after generation 30. A
...         # mendelianGenoTransmitter defined in randomMating is responsible
...         # for genotype transmission before that.
...         recombinator(rates=0.01, begin=30),
...         stat(LD=[0, 1]),
...         pyEval(r"'gen %d, LD: %.2f\n' % (gen, LD[0][1])", step=20)
...     ],
...     gen=100
... )
gen 0, LD: 0.25
gen 20, LD: 0.25
gen 40, LD: 0.22
gen 60, LD: 0.18
gen 80, LD: 0.14
(100,)
>>>
```

Section 2.8.4 and 3.4 list all genotype transmitters, Section 2.8.5 demonstrates how to define your own genotype transmitter.

2.6.5 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 2.32 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 2.32: 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: 76
1: 79
2: 78
3: 79
4: 79
(5,)
>>>
```

2.6.6 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.** This is essentially how operator `terminateIf` works.

Example 2.33 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 2.6.8 for details).

Example 2.33: A frequency dependent mutation operator

```
    # 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, k=2, rates=mu1, loci=[i])
        else:
            KamMutate(pop, k=2, rates=mu2, loci=[i])
    return True

simu = simulator(population(size=10000, loci=[2, 3]),
    randomMating())
simu.evolve(
    preOps = [
```

Example 2.53 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 2.34: 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
0.11 0.20 0.12 0.20 0.11
0.18 0.21 0.19 0.21 0.18
0.19 0.21 0.20 0.21 0.20
(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 2.35 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 2.35: 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, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 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.

2.6.7 Define your own operators **

`pyOperator` is a Python class so you can derive your own operator from this operator. The tricky part is that the constructor of the derived operator needs to call the `__init__` function of `pyOperator` will proper functions. This technique has been used by simuPOP in a number of occasions. For example, the `varPlotter` operator defined in `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 2.53 rewrites the `dynaMutator` defined in Example 2.33 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 2.36: 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, k=2, rates=self.mu1, loci=[i])
...             else:
...                 KamMutate(pop, k=2, rates=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 0.21 0.02
0.12 0.22 0.10 0.21 0.10
0.19 0.23 0.17 0.22 0.17
0.21 0.24 0.20 0.20 0.21
(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.

2.6.8 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 2.32 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 2.37: 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. **Note:** Whereas output files specified by `'>'` are closed immediately after they are written, those specified by `'>>'` and `'>>>'` are not closed after the operator is applied to a population. This is not a problem when operators are used in a simulator because `simulator.evolve` closes all files opened by operators, but can cause trouble when the operator is applied directly to a population. For example, multiple calls to `Dump(pop, output='>>file')` will dump pop to `file` repeatedly but `file` will not be closed afterward. In this case, `CloseOutput('file')` should be used to explicitly close the file.

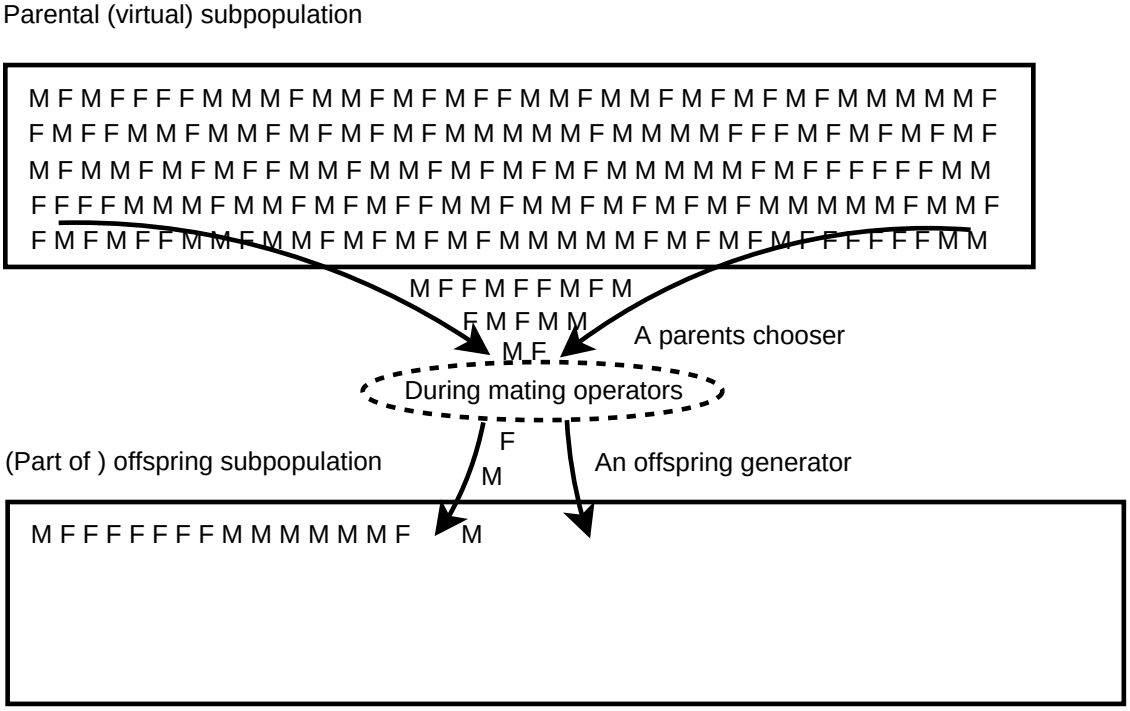
2.7 Mating Schemes

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 2.38 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 2.38: 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
... )
[795, 705]
[936, 564]
[942, 558]
(3,)
>>>
```

Example 2.39 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 2.39: 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
... )
[781, 719]
[500, 1000]
```

```
[820, 680]
[500, 1000]
[808, 692]
[500, 1000]
(3,)
>>>
```

Example 2.44 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 2.40: 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 2.41 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 2.41: 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))
[7, 5, 8, 8, 2]
>>> # Case 3: A geometric distribution
>>> checkNumOffspring(randomMating(numOffspring=(GeometricDistribution, 0.3)))
[1, 1, 3, 3, 2, 2, 2, 3, 4, 2, 1, 1, 1, 2, 2]
>>> # Case 4: A Possition distribution
>>> checkNumOffspring(randomMating(numOffspring=(PoissonDistribution, 3)))
[4, 4, 3, 5, 3, 4, 5, 2]
>>> # Case 5: A Binomial distribution
>>> checkNumOffspring(randomMating(numOffspring=(BinomialDistribution, 0.1, 10)))
[1, 2, 2, 4, 3, 1, 1, 3, 3, 2, 2, 2, 1, 3]
>>> # Case 6: A uniform distribution
>>> checkNumOffspring(randomMating(numOffspring=(UniformDistribution, 2, 6)))
[5, 4, 5, 6, 3, 5, 2]
>>>
```

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 2.42 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 2.42: 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))
'FFFFFFMMFFFFMMMMFFMMFFFMMMFMFMMFMFFFMMFM'
>>> # Case 3: ProbOfMale (Specify probability of male)
>>> checkSexMode(randomMating(sexMode=(ProbOfMale, 0.8)))
'MMMMMMMMMMMMMMMMMMFMFFFFMFMMMMMFMMFFMMMM'
>>> # Case 4: NumOfMale (Specify number of male in each family)
>>> checkSexMode(randomMating(numOffspring=3, sexMode=(NumOfMale, 1)))
'MFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFMFFM'
>>> # Case 5: NumOfFemale (Specify number of female in each family)
>>> checkSexMode(randomMating(
...     numOffspring=(UniformDistribution, 4, 6),
...     sexMode=(NumOfFemale, 2))
... )
'FFMMFFMMMFFMMMMFFMMMMFFMMMFFMMFFMMMMFFMM'
>>>
```

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 2.43 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 2.43: 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 2.44 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 2.44: 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]
[12, 12, 87, 87, 29, 29, 85, 85, 88, 88, 56, 56, 47, 47, 48, 48, 56, 56, 24, 24]
>>> [ind.intInfo('mother_idx') for ind in pop.individuals()][:20]
[6, 7, 52, 63, 41, 99, 41, 52, 0, 9, 3, 4, 96, 1, 18, 63, 9, 6, 7, 51]
>>>
```

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 2.45 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 2.45: 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 [0, 1, 1, 0, 1, 0, 1, 1, 0, 1]
>>> print par.sex(), par.genotype()
2 [0, 1, 1, 0, 1, 0, 1, 1, 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 2.46 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 2.46: 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,
... )
0.51
0.93
0.69
0.82
(20,)
>>> pop = simu.extract(0)
>>> [ind.intInfo('father_idx') for ind in pop.individuals()][:10]
[60, 60, 156, 60, 60, 156, 156, 156, 156, 60]
>>> [ind.intInfo('mother_idx') for ind in pop.individuals()][:10]
[11, 223, 655, 896, 340, 701, 525, 249, 407, 727]
>>>
```

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 2.47 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 2.47: 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 (), 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: MU 00000 00000 | _____ _____ |  0 0
   5: FU 11111 11111 | 22222 22222 |  0 0
   6: MU 00000 00000 | _____ _____ |  0 0
   7: MU 00000 00000 | _____ _____ |  0 0
   8: MU 00000 00000 | _____ _____ |  0 0
   9: MU 00000 00000 | _____ _____ |  0 0
Subpopulation 0 (), 10 individuals:
   0: MU 22222 11111 | _____ _____ |  6 5
   1: MU 22222 22222 | _____ _____ |  4 5
   2: FU 22222 22222 | 00000 00000 |  4 1
   3: MU 22222 11111 | _____ _____ |  8 5
   4: MU 11111 22222 | _____ _____ |  4 5
   5: FU 11111 22222 | 00000 00000 |  8 5
   6: FU 22222 11111 | 00000 00000 |  8 1
   7: MU 22222 22222 | _____ _____ |  2 5
   8: MU 22222 22222 | _____ _____ |  6 5
   9: FU 22222 11111 | 00000 00000 |  4 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 2.48 initializes each chromosome with different alleles to demonstrate how these alleles are transmitted in this population.

Example 2.48: 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(rates=0.1)],
...     gen = 1
... )
(1,)
>>> Dump(simu.population(0), width=3, structure=False, max=10)
Subpopulation 0 (), 20 individuals:
   0: MU   7  7  7  7  7  7  7  7 |   7  7  7  7  7  7  7  7
   1: MU  16 16 16 16 16 16 16 16 |  16 16 16 16 16 17 17 17
   2: FU   4  4  5  5  5  5  5  5 |   5  5  5  5  4  4  4  4
   3: MU  18 18 18 18 18 19 19 18 |  18 18 18 18 19 19 19 18
   4: MU   7  7  7  7  6  6  6  6 |   7  7  6  6  6  6  6  6
   5: FU   8  8  8  8  8  8  8  8 |   9  8  9  9  9  9  9  9
   6: FU  18 19 19 19 18 18 18 18 |  19 19 19 19 19 19 19 19
   7: MU  39 39 39 39 39 39 39 39 |  39 39 39 39 39 39 39 38
   8: MU  38 38 39 39 39 39 39 39 |  39 39 39 39 38 38 38 38
   9: FU   9  8  8  8  8  8  8  8 |   8  8  8  8  8  8  8  8
>>>
```

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.

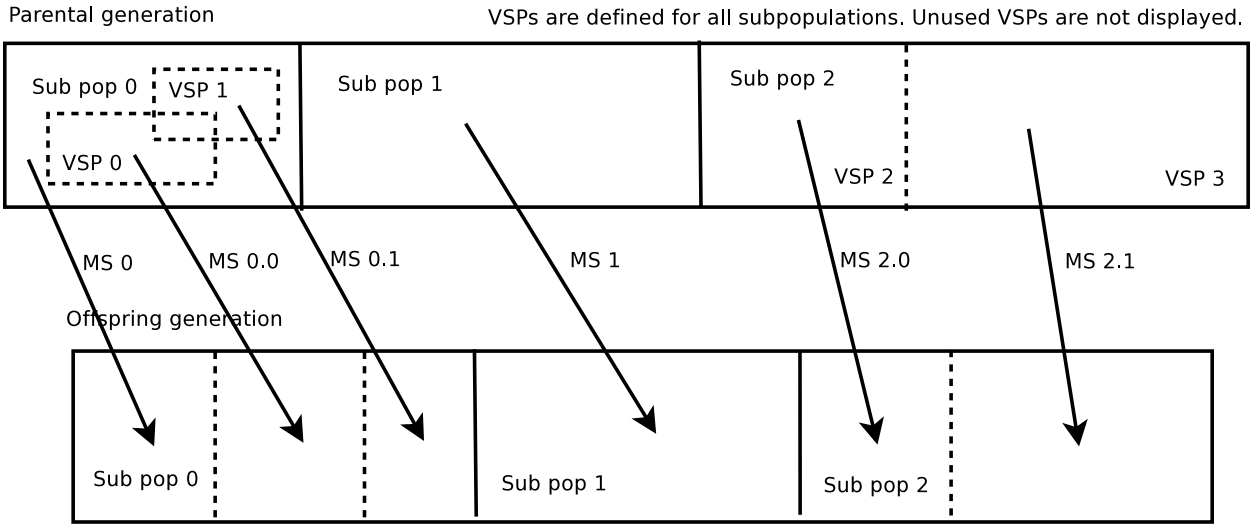
A heterogeneous mating scheme (`heteroMating`) accepts a list of mating schemes that are applied to different subpopulation or virtual subpopulations. If multiple mating schemes are applied to the same subpopulation, each of them only population part of the offspring subpopulation. This is illustrated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

For example, Example 2.49 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 2.49: 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]
[31, 31, 194, 194, 481, 481, 516, 516, 237, 237]
>>> [ind.intInfo('father_idx') for ind in pop.individuals(1)][:10]
[1811, 1811, 1811, 1811, 1687, 1687, 1687, 1687, 1172, 1172]
```

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 2.50 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 2.50: 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]
[38, 82, 50, 86, 26, 2, 80, 91, 37, 81, 128, 169, 192, 193, 6]
>>> [ind.intInfo('mother_idx') for ind in pop.individuals(0)][:15]
[34, 81, 180, -1, -1, -1, 17, 35, -1, -1, 69, 79, 34, 173, -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 2.51 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 2.51: 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 *

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 2.52 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 2.52: Define a random mating scheme

```
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 2.53.

Example 2.53: 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 (), 100 individuals:
   0: MU 432412203434432 | 230200210200441 |  0
   1: FU 312343131104411 | 011022122030121 |  0
   2: MU 441234442310144 | 241203011302214 |  0
   3: FU 342033213313220 | 131401134443001 |  0
   4: MU 220002103203333 | 324012301404443 |  0
Subpopulation 0 (), 100 individuals:
   0: FU 432412203434432 | 230200210200441 |  0
   1: FU 312343131104411 | 011022122030121 |  1
   2: FU 441234442310144 | 241203011302214 |  2
   3: FU 131401134443001 | 342033213313220 |  3
   4: FU 324012301404443 | 324012301404443 |  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 2.52 uses a `mendelianGenoTransmitter` as the default genotype transmitter. Additional during-mating operators can be added to the operator list, as shown in Example 2.51, 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 2.54 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 2.54: 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    0.51
0.51    0.51
0.52    0.53
0.53    0.53
0.54    0.55
(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.

Only one genotype transmitter will be applied to an offspring. If another transmitter is used in the `evolve` function, it will override the transmitter defined in the mating scheme. 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 during-mating operators.

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 2.55 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 2.55: 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(rates=0.1),
...         parentsTagger(),
...         dumper(structure=False),
...     ],
...     gen = 2
... )
Subpopulation 0 (), 10 individuals:
   0: FU 10003 30132 _____ 12322 12322 | 32313 23302 _____ 00000 00000 |  6 0
   1: FU 00133 00000 _____ 21030 10030 | 32002 30131 _____ 00000 00000 |  1 5
   2: FU 23103 30020 _____ 12013 12013 | 13330 23302 _____ 00000 00000 |  6 4
   3: MU 32120 20001 _____ 10030 10030 | 10023 _____ 12101 00000 00000 |  1 5
   4: FU 31300 00103 _____ 20003 10330 | 32002 30131 _____ 00000 00000 |  1 8
   5: MU 10032 00030 _____ 00022 01102 | 32002 _____ 12101 00000 00000 |  1 2
   6: FU 31300 20303 _____ 20003 10330 | 13330 23302 _____ 00000 00000 |  6 8
   7: MU 31013 00100 _____ 10330 20003 | 32003 _____ 12101 00000 00000 |  1 8
   8: FU 02130 00001 _____ 10030 21030 | 32002 30131 _____ 00000 00000 |  1 5
   9: FU 10031 31122 _____ 01102 00022 | 13330 23302 _____ 00000 00000 |  6 2
Subpopulation 0 (), 10 individuals:
   0: MU 13331 23302 _____ 01102 01102 | 32002 _____ 12101 00000 00000 |  5 9
   1: MU 32002 30131 _____ 20003 20003 | 12002 _____ 12101 00000 00000 |  5 4
   2: MU 13330 23302 _____ 12013 12013 | 32002 _____ 12101 00000 00000 |  5 2
   3: FU 10031 31122 _____ 01102 01102 | 31013 00100 _____ 00000 00000 |  7 9
   4: FU 32002 30131 _____ 10330 20003 | 32003 00100 _____ 00000 00000 |  7 4
   5: FU 10031 23302 _____ 01102 00022 | 32002 00030 _____ 00000 00000 |  5 9
   6: MU 10330 31302 _____ 00022 01102 | 10023 _____ 12101 00000 00000 |  3 9
   7: MU 13303 20020 _____ 12013 12013 | 10032 _____ 12101 00000 00000 |  5 2
   8: FU 10031 31122 _____ 00022 00022 | 32032 00030 _____ 00000 00000 |  5 9
   9: FU 31000 00103 _____ 10330 20003 | 32120 20001 _____ 00000 00000 |  3 4
(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 2.33), it is much better to derive an operator from pyOperator, as the case in Example 2.36.

Example 2.57 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`. It is important to set `transmitter=True` so that this operator will be treated as a genotype transmitter.

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 2.56 outputs the population in two generations. You should notice that paternal chromosome are not recombined when they are

transmitted to offspring.

Example 2.56: A customized genotype transmitter for sex-specific recombination

```
>>> class sexSpecificRecombinator(pyOperator):
...     def __init__(self, intensity=0, rates=0, loci=[], convMode=NoConversion,
...             maleIntensity=0, maleRates=0, maleLoci=[], maleConvMode=NoConversion,
...             *args, **kwargs):
...         # This operator is used to recombine maternal chromosomes
...         self.recombinator = recombinator(rates, intensity, loci, convMode)
...         # This operator is used to recombine paternal chromosomes
...         self.maleRecombinator = recombinator(maleRates, maleIntensity,
...             maleLoci, maleConvMode)
...         #
...         self.initialized = False
...         # Note the use of parameter isTransmitter
...         pyOperator.__init__(self, func=self.transmitGenotype,
...             stage=DuringMating, isTransmitter=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(rates=0.1, maleRates=0),
...         dumper(structure=False),
...     ],
...     gen = 2
... )
Subpopulation 0 (), 10 individuals:
   0: MU 311020330302321 201032310220302 | 010221122003230 030333312112112 |  1 2
   1: MU 320002101012120 201210221100211 | 021110202100000 220121022300033 |  5 3
   2: MU 020023230213200 101101023112001 | 120033323211121 301112000033313 |  5 0
   3: FU 310130200030232 003030000032203 | 323203323302000 202230220320023 |  7 4
   4: FU 311030200300000 111110330031301 | 100210003000032 100300302232033 |  9 4
   5: FU 303120300010130 000320010200100 | 010221122003230 030333312112112 |  1 8
   6: MU 230022300010223 000320010200102 | 120033323211121 301112000033313 |  5 8
   7: FU 220023230203110 101101023112001 | 323203323302000 202230220320023 |  7 0
   8: MU 230022010202130 000311230200100 | 100210003000032 130133320202230 |  9 8
   9: FU 001022121203203 232021200023201 | 120033323211121 220121022300033 |  5 6
Subpopulation 0 (), 10 individuals:
   0: MU 323203323302000 003030000320023 | 010221122003230 030333312112112 |  0 3
   1: MU 220023230203110 101130220320023 | 010221122003230 201032310220302 |  0 7
   2: FU 100210000300032 100300302232033 | 311020330302321 201032310220302 |  0 4
   3: MU 303120300010130 030333312112112 | 021110202100000 201210221100211 |  1 5
   4: FU 220223323302010 101101220320023 | 230022300010223 301112000033313 |  6 7
   5: MU 120033323211121 220121022300033 | 230022300010223 000320010200102 |  6 9
```

```
   6: FU 323203300032232 203030000032203 | 100210003000032 000311230200100 |  8 3
   7: FU 001022121203203 232021200023201 | 311020330302321 201032310220302 |  0 9
   8: MU 220023230203110 101101023112023 | 230022300010223 000320010200102 |  6 7
   9: FU 313203323330232 202030000032203 | 020023230213200 301112000033313 |  2 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 2.53 and 2.54. 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 2.57 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 2.57: 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 (), 100 individuals:
   0: MU 1101111011 | 1111111111 |  14 15 0
   1: FU 1101111011 | 0101111111 |  14 15 0
   2: MU 1111111110 | 1110111001 |  58 59 0
   3: FU 1111111110 | 1110111001 |  58 59 0
   4: MU 1100001101 | 1111111111 |  42 43 0
   5: FU 1110111111 | 1111101101 |  42 43 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 2.58 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 2.58: 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 (*references to individual objects or indexes relative to a subpopulation*). Note that `pyParentsChooser` does not support virtual subpopulation but you can mimic the effect by returning only parents from certain virtual subpopulations.

Example 2.59 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 2.59: 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 pop.individuals(sp) \
...             if x.sex() == Male and x.info('rank') == rank])
...         females.append([x for x in pop.individuals(sp) \
...             if x.sex() == Female and x.info('rank') == rank])
...     while True:
...         # choose a rank randomly
...         rank = pop.individual(randint(0, pop.subPopSize(sp) - 1), sp).intInfo('rank')
```

```
...         yield males[rank][randint(0, len(males[rank]) - 1)], \
...             females[rank][randint(0, len(females[rank]) - 1)]
...
>>> 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 2.60 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 2.60: 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 2.61 lists an interface file for the C++ class defined in Example 2.60. Please refer to the SWIG reference manual for details.

Example 2.61: 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 2.62.

Example 2.62: 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 2.60 demonstrates how to use such a C++ parents chooser in your simuPOP script. It uses the same Python parent chooser interface as in 2.59, but leaves all the (potentially) computationally intensive parts to the C++ level `myParentsChooser` object.

Example 2.63: 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 2.64 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 `evolve()` call.

Example 2.64: 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('len(alleleNum[5]) == 1'),
...     ],
... )
Allele 0 is lost in rep 1 at gen 37
Allele 0 is fixed in rep 0 at gen 76
Allele 0 is lost in rep 2 at gen 154
(72, 33, 150)
>>> simu.gen()
155
>>>
```

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 2.65 shows the operator calling sequence for Example 2.27.

Example 2.65: 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 2.66: 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 2.67 demonstrates the implementation of such a scenario.

Example 2.67: 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()]
[314, 314, 272, 272, 27, 27, 103, 103, 409, 409]
>>>
```

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 2.68 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 2.68: 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()
...
5 (1149.0, 2057.0, 2577.0, 2923.0, 7904.0)
5 (523.0, 1659.0, 2321.0, 3913.0, 9832.0)
4 (1126.0, 5003.0, 9343.0, 9916.0)
>>>
```

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, vars='numOfMale_sp')
>>> print pop.dvars(0).numOfMale
295
>>> 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 (), 2 individuals:
   0: MU 01011 0011111 | 01110 1111111
   1: FU 11111 1111110 | 01101 1111111
Subpopulation 1 (), 3 individuals:
   2: MU 00000 0000000 | 10001 1000000
   3: MU 00000 0100000 | 11000 0001001
   4: MU 00100 0000010 | 00000 0000000
>>>
```

It is sometimes desired to create 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 (), 2 individuals:
   0: MU 10111 1111111 | 11011 1111101
   1: MU 10111 1111111 | 11011 1111101
Subpopulation 1 (), 3 individuals:
   2: MU 10111 1110111 | 11011 1111111
   3: MU 10111 1110111 | 11011 1111111
   4: MU 10111 1110111 | 11011 1111111
>>>
```

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 (), 2 individuals:
   0: FU 11111 2222222 | 33333 4444444
```

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

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 (), 6 individuals:
   0: FU 01234 0000000 | 56789 3333333
   1: MU 01234 2222222 | 56789 2222222
   2: MU 01234 2222222 | 56789 2222222
   3: FU 01234 0000000 | 56789 4444444
   4: MU 01234 2222222 | 56789 2222222
   5: FU 01234 0000000 | 56789 4444444
Subpopulation 1 (), 8 individuals:
   6: MU 01234 2222222 | 56789 2222222
   7: MU 01234 2222222 | 56789 2222222
   8: FU 01234 0000000 | 56789 3333333
   9: MU 01234 2222222 | 56789 2222222
  10: MU 01234 2222222 | 56789 2222222
  11: MU 01234 2222222 | 56789 2222222
  12: MU 01234 2222222 | 56789 2222222
  13: MU 01234 2222222 | 56789 2222222
>>>
```

3.2 Expressions and statements

3.2.1 Output a Python string (operator `pyOutput`)

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

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])
0.810, 0.800, 0.790, 0.815, 0.820
>>>
```

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, 2.21, 2.27 and 2.29) 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 437, #mother 417
gen 1, #father 433, #mother 437
gen 2, #father 430, #mother 449
(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
2.02.02.0
>>> InfoEval(pop, 'ind.allele(0, 0)', exposeInd='ind');print
10110010111101111111
>>> # use population variables
>>> pop.dvars().b = 5
```

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

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, 2.0, 1.0, 1.0, 1.0, 2.0, 2.0, 2.0, 2.0)
>>> InfoExec(pop, 'c=a+b')
>>> print pop.indInfo('c')[:10]
(2.0, 3.0, 3.0, 2.0, 2.0, 2.0, 3.0, 3.0, 3.0, 3.0)
>>> pop.dvars().d = 5
>>> InfoExec(pop, 'c+=d', usePopVars=True)
>>> print pop.indInfo('c')[:10]
(7.0, 8.0, 8.0, 7.0, 7.0, 7.0, 8.0, 8.0, 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
... )
[795, 1115, 1090]
[636, 1184, 1180]
[482, 1229, 1289]
[405, 1299, 1296]
[320, 1339, 1341]
(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, a few functions are defined in module `simuUtil`.

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

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

  for a traditional island model where individuals have equal probability of migrating to any other subpopulations.

- `MigrHierarchicalIslandRates(r1, r2, n)` models a hierarchical island model in which local populations are grouped into neighborhoods within which there is considerable gene flow and between which there is less gene flow. n should be a list of group size. r_1 is the within-group migration rate and r_2 is the cross-group migration rate. That is to say, an individual in an island has probability $1-r_1-r_2$ to say, r_1 to be a migratant to other islands in the group (migration rate depending on the size of group), and r_2 to be a migrant to other islands in another group (migration rate depending on the number of islands in other groups). Both r_1 and r_2 can vary across groups of islands. For example, `MigrHierarchicalIslandRates([r11, r12], r2, [3, 2])` returns a 5×5 migration matrix

$$\begin{pmatrix} 1-r_{11}-r_2 & \frac{r_{11}}{2} & \frac{r_{11}}{2} & \frac{r_2}{2} & \frac{r_2}{2} \\ \frac{r_{11}}{2} & 1-r_{11}-r_2 & \frac{r_{11}}{2} & \frac{r_2}{2} & \frac{r_2}{2} \\ \frac{r_{11}}{2} & \frac{r_{11}}{2} & 1-r_{11}-r_2 & \frac{r_2}{2} & \frac{r_2}{2} \\ \frac{r_2}{3} & \frac{r_2}{3} & \frac{r_2}{3} & 1-r_{12}-r_2 & r_{12} \\ \frac{r_2}{3} & \frac{r_2}{3} & \frac{r_2}{3} & r_{12} & 1-r_{12}-r_2 \end{pmatrix}$$

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

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

  and if `circular=True`, returns

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

Many more migration models have been proposed and studied, sometimes under different names with slightly different definitions. Please refer to the *operators* section of the simuPOP online cookbook for the implementation of more migration models. If you cannot find your model there, it should not be too dificult to construct a migration rate matrix for it and please consider adding your model to the cookbook.

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, vars='numOfMale_sp'),
...         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
461/925 549/1075
415/856 595/1144
417/856 583/1144
(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, namely 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()
(240, 266, 238, 256, 1000, 240, 266, 238, 256)
>>> 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 Genotype transmitters

3.4.1 Generic genotype transmitters (operators `genoTransmitter`, `cloneGenoTransmitter`, `mendelianGenoTransmitter`, `selfingGenoTransmitter`, `haplodiploidGenoTransmitter`, and `mitochondrialGenoTransmitter`) *

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

- `genoTransmitter`: This genotype transmitter is usually used by customized genotype transmitters because it provides some utility functions that are more efficient than their Pythonic counterparts.
- `cloneGenoTransmitter`: Copy all genotype from a parent to an offspring. It also copies parental sex to the offspring because sex can be genotype determined. This genotype transmitter is used by mating scheme `cloneMating`. This genotype transmitter can be applied to populations of **any ploidy** type.
- `mendelianGenoTransmitter`: Copy genotypes from two parents (a Male and a Female) to an offspring following Mendel's laws, used by mating scheme `randomMating`.This genotype transmitter can only be applied to **diploid** populations.
- `selfingGenoTransmitter`: Copy genotypes from one parent to an offspring using self-fertilization, used by mating scheme `selfMating`. This genotype transmitter can only be applied to **diploid** populations.

- `haplodiploidGenoTransmitter`: Set genotype to male and female offspring differently in a haplodiploid population, used by mating scheme `haplodiploidMating`. This genotype transmitter can only be applied to **haplodiploid** populations.
- `mitochondrialGenoTransmitter`: Treat all customized chromosomes or specified chromosomes as mitochondrial chromosomes and transmit maternal mitochondrial chromosomes randomly to an offspring. This genotype transmitter can be applied to populations of **any ploidy** type. It trasmits the first homologous copy of chromosomes maternally and clears alleles on other homologous copies of chromosomes of an offspring. **This operator is not a genotype transmitter** so that its existence will not block the application of another genotype transmitter.

3.4.2 Recombination (Operator `recombinator`)

The generic genotype transmitters do not handle genetic recombination. A genotype transmitter `recombinator` is provided for such purposes, and can be used with `randomMating` and `selfMating` (replace `mendelianGenoTransmitter` and `selfingGenoTransmitter` used in these mating schemes).

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

Recombination rates could be specified in the following ways:

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

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

   recombines a chromosome only after loci 2 (between 2 and 3) and 5 (between 5 and 6).
3. If parameter `loci` is given, different recombination rate can be given to each of them. For example

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

   uses two different recombination rates after loci 2 and 5.
4. If recombination rates vary across your chromosomes, a long list of `rate` and `loci` may be needed to specify recombination rates one by one. An alternative method is to specify a **recombination intensity**. Recombination rate between two adjacent loci is calculated as the product of this intensity and distance between them. For example, if you apply operator

   ```
   recombinator(intensity=0.1)
   ```

   to a population

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

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

Example 3.19: Genetic recombination at all and selected loci

```
>>> simu = simulator(population(size=[1000], loci=[100]),
...     randomMating(), rep=2)
>>> simu.evolve(
...     preOps = [initByValue([0]*100 + [1]*100)],
...     ops = [
...         recombinator(rates=0.01, reps=0),
...         recombinator(rates=[0.01]*10, loci=range(50, 60), reps=1),
```

```
...         stat(LD=[[40, 55], [60, 70]]),
...         pyEval(r'"%d:\t%.3f\t%.3f\t" % (rep, LD_prime[40][55], LD_prime[60][70])'),
...         pyOutput('\n', reps=-1)
...     ],
...     gen = 5
... )
0:      0.736   0.817   1:      0.915   1.000
0:      0.597   0.738   1:      0.882   1.000
0:      0.509   0.646   1:      0.842   1.000
0:      0.451   0.608   1:      0.811   1.000
0:      0.450   0.558   1:      0.796   1.000
(5, 5)
>>>
```

Example 3.19 demonstrates how to specify recombination rates for all loci or for specified loci. In this example, two replicates of a population are evolved, subject to two different recombinators. The first recombinator applies the same recombination rate between all adjacent loci, and the second recombinator recombines only after loci 50 - 59. Because there is no recombination event between loci 60 and 70 for the second replicate, linkage disequilibrium values between these two loci does not decrease as what happens in the first replicate.

Example 3.20: Genetic recombination rates specified by intensity

```
>>> simu = simulator(population(size=[1000], loci=3, lociPos=[0, 1, 1.1]),
...     randomMating())
>>> simu.evolve(
...     preOps = [initByValue([0]*3 + [1]*3)],
...     ops = [
...         recombinator(intensity=0.01),
...         stat(LD=[[0, 1], [1, 2]]),
...         pyEval(r'"%.3f\t%.3f\n" % (LD_prime[0][1], LD_prime[1][2])', step=10)
...     ],
...     gen = 50
... )
0.984   1.000
0.869   0.998
0.758   1.000
0.753   0.992
0.712   0.986
(50,)
>>>
```

Example 3.20 demonstrates the use of the `intensity` parameter. In this example, the distances between the first two loci and the latter two loci are 1 and 0.1 respectively. This leads recombination rates 0.01 and 0.001 respectively with a recombination intensity 0.01. Consequently, LD between the first two loci decay much faster than the latter two.

If more advanced recombination model is desired, a customized genotype transmitter can be used. For example, Example 2.56 uses two recombinators to implement sex-specific recombination. **Note:** Both loci positions and recombination intensity are unitless. You can assume different unit for loci position and recombination intensity as long as the resulting recombination rate makes sense.

3.4.3 Gene conversion (Operator `recombinator`) *

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

- Two (out of four) chromatids pair and a single strand cut is made in each chromatid
- Strand exchange takes place between the chromatids

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

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

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

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

Note that

- If tract length is determined by length (`TractLength` or `ExponentialDistribution`), the starting point of the flanking region is uniformly distributed between marker $i-1$ and i, if the recombination happens at marker i. That is to say, it is possible that no marker is converted with a positive tract length.
- A conversion event will act like a recombination event if its flanking region exceeds the end of a chromosome, or if another recombination event happens before the end of the flanking region.

Example 3.21 compares two recombinators. The first recombinator is a regular recombinator that recombine between loci 50 and 51. The second recombinator is a conversion operator because every recombination event will become a conversion event (prob=1). Because a second recombination event will surely happen between loci 60 and 61, there will be either no or double recombination events between loci 40, 70. LD between these two loci therefore does not decrease, although LD between locus 55 and these two loci will decay.

Example 3.21: Gene conversion

```
>>> simu = simulator(population(size=[1000], loci=[100]),
...     randomMating(), rep=2)
>>> simu.evolve(
...     preOps = [initByValue([0]*100 + [1]*100)],
...     ops = [
...         recombinator(rates=0.01, loci=50, reps=0),
...         recombinator(rates=0.01, loci=50, reps=1,
...             convMode=(NumMarkers, 1, 10)),
...         stat(LD=[[40, 55], [40, 70]]),
...         pyEval(r'"%d:\t%.3f\t%.3f\t" % (rep, LD_prime[40][55], LD_prime[40][70])'),
...         pyOutput('\n', reps=-1)
...     ],
...     gen = 5
... )
```

```
0:      0.982   0.982   1:      0.982   1.000
0:      0.978   0.978   1:      0.976   1.000
0:      0.960   0.960   1:      0.972   1.000
0:      0.948   0.948   1:      0.964   1.000
0:      0.938   0.938   1:      0.952   1.000
(5, 5)
>>>
```

3.5 Mutation

A mutator (a mutation operator) mutates alleles at certain loci from one allele to another. Because alleles are simple non-nagative numbers that can be intrepreted as nucleotides, codons, squences of nucleotides or even genetic deletions, appropriate mutation models have to be chosen for different types of loci. Please refer to Section 2.3 for a few examples.

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

3.5.1 Mutation models specified by rate matrixes (`matrixMutator`)

A mutation model can be defined as a **mutation rate matrix** $(p_{ij})_{n\times n}$ where p_{ij} is the probability that an allele i mutates to j per generation per locus. Although mathematical formulation of p_{ij} are sometimes unscaled, simuPOP assumes $\sum_{j=0}^{n-1} p_{ij} = 1$ for all i and requires such rate matrixes in the specification of a mutation model. p_{ii} of such a matrix are ignored because they are automatically calculated from $p_{ii} = 1 - \sum_{j\neq i} p_{ij}$.

A `matrixMutator` is defined to mutate between alleles 0, 1, ..., $n-1$ according to a given rate matrix. Conceptually speaking, this mutator goes through each mutable allele and mutates it to allele $0, 1, .., n-1$ according to probabilities p_{ij}, $j = 0, ..., n-1$. Most alleles will be kept intact because mutations usually happen at low probability (with p_{ii} close to 1). For example, Example 3.22 simulates a locus with 3 alleles. Because the rate at which allele 2 mutats to alleles 0 and 1 is higher than the rate alleles 0 and 2 mutate to allele 2, the frequency of allele 2 decreases over time.

Example 3.22: General mutator specified by a mutation rate matrix

```
>>> simu = simulator(population(size=[2000], loci=1),
...     randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([0.2, 0.3, 0.5])],
...     ops = [
...         matrixMutator(rate = [
...             [0, 1e-5, 1e-5],
...             [1e-4, 0, 1e-4],
...             [1e-3, 1e-3, 0]
...         ]),
...         stat(alleleFreq=0, step=100),
...         pyEval(r"', '.join(['%.3f' % alleleFreq[0][x] for x in range(3)]) + '\n'",
...             step=100),
...     ],
...     gen=1000
... )
0.201, 0.295, 0.503
0.210, 0.303, 0.486
```

```
0.257, 0.372, 0.370
0.273, 0.573, 0.154
0.276, 0.540, 0.184
0.264, 0.556, 0.180
0.245, 0.625, 0.130
0.225, 0.692, 0.083
0.254, 0.629, 0.117
0.324, 0.619, 0.057
(1000,)
>>>
```

3.5.2 k-allele mutation model (`kamMutator`)

A k-allele model assumes k alleles $(0, ..., k-1)$ at a locus and mutate between them using rate matrix

$$p_{ij} = \begin{pmatrix} 1-\mu & \frac{\mu}{k-1} & \cdots & \frac{\mu}{k-1} \\ \frac{\mu}{k-1} & 1-\mu & \cdots & \frac{\mu}{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mu}{k-1} & \frac{\mu}{k-1} & \cdots & 1-\mu \end{pmatrix}$$

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

This mutation model is a special case of the `matrixMutator` but a specialized `kamMutator` is recommended because it provides better performance, especially when k is large. In addition, this operator allows different mutation rates at different loci. When k is not specified, it is assumed to be the number of allowed alleles (e.g. 2 for binary modules). Example 3.23 desmonstrates the use of this operator where parameters `rate` and `loci` are used to specify different mutation rates for different loci. Because this operator treats all alleles equally, all alleles will have the same allele frequency in the long run.

Example 3.23: A k-allele mutation model

```
>>> pop = population(size=[2000], loci=1*3)
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = initSex(),
...     ops = [
...         kamMutator(k=5, rates=[1e-2, 1e-3], loci=[0, 1]),
...         stat(alleleFreq=range(3), step=100),
...         pyEval(r"', '.join(['%.3f' % alleleFreq[x][0] for x in range(3)]) + '\n'",
...             step=100),
...     ],
...     gen=500
... )
0.990, 0.999, 1.000
0.449, 0.930, 1.000
0.265, 0.843, 1.000
0.251, 0.833, 1.000
0.247, 0.757, 1.000
(500,)
>>>
```

3.5.3 Diallelic mutation models (`snpMutator`)

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

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

$$R = \begin{pmatrix} 1-u & u \\ v & 1-v \end{pmatrix}.$$

Despite of its name, this mutator can be used in many theoretical models assuming $\Pr(A \to a) = u$ and $\Pr(a \to A) = v$. If $v = 0$, mutations will be directional. Example 3.24 applies such a directional mutaton model to two loci, but with a purifying selection applied to the first locus. Because of the selection pressure, the frequency of allele 1 at the first locus does not increase indefinitely as allele 1 at the second locus.

Example 3.24: A diallelic directional mutation model

```
>>> pop = population(size=[2000], loci=[1, 1], infoFields='fitness')
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = initSex(),
...     ops = [
...         snpMutator(u=0.001),
...         maSelector(loci=0, fitness=[1, 0.99, 0.98]),
...         stat(alleleFreq=[0, 1], step=100),
...         pyEval(r"'%.3f\t%.3f\n' % (alleleFreq[0][1], alleleFreq[1][1])",
...             step=100),
...     ],
...     gen=500
... )
0.001   0.001
0.042   0.074
0.057   0.085
0.068   0.189
0.077   0.404
(500,)
>>>
```

3.5.4 Nucleotide mutation models (`acgtMutator`)

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

$$\begin{array}{c|cccc} & A & C & G & T \\ A & - & x_1 & x_2 & x_3 \\ C & x_4 & - & x_5 & x_6 \\ G & x_7 & x_8 & - & x_9 \\ T & x_{10} & x_{11} & x_{12} & - \end{array}$$

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

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

specifies a nucleotide mutator using Kimura's 2-parameter model with $\mu = 10^{-5}$ and $\kappa = 0.5$. Because multiple parameters could be involved for a particular mutation model, **the definition of a mutation rate and other paramters are model dependent and may varying with different mathematical representation of the models**.

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

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

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

$$R = \begin{pmatrix} - & \frac{\mu}{4} & \frac{\mu}{4} & \frac{\mu}{4} \\ \frac{\mu}{4} & - & \frac{\mu}{4} & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4} & - & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4} & \frac{\mu}{4} & - \end{pmatrix}$$

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

   Kimura 's model distinguishes transitions ($A \longleftrightarrow G$, and $C \leftrightarrow T$ namely $0 \longleftrightarrow 2$ and $1 \longleftrightarrow 3$ with probability $\frac{\mu}{4}\kappa$) and transversions (others) with probability $\frac{\mu}{4}$. It would be a Jukes and Cantor model if $\kappa = 1$.

$$R = \begin{pmatrix} - & \frac{\mu}{4} & \frac{\mu}{4}\kappa & \frac{\mu}{4} \\ \frac{\mu}{4} & - & \frac{\mu}{4} & \frac{\mu}{4}\kappa \\ \frac{\mu}{4}\kappa & \frac{\mu}{4} & - & \frac{\mu}{4} \\ \frac{\mu}{4} & \frac{\mu}{4}\kappa & \frac{\mu}{4} & - \end{pmatrix}$$

3. Felsenstein 1981 model: model='F81', rate=[μ, π_A, π_C, π_G].

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

$$R = \begin{pmatrix} - & \mu\pi_C & \mu\pi_G & \mu\pi_T \\ \mu\pi_A & - & \mu\pi_G & \mu\pi_T \\ \mu\pi_A & \mu\pi_C & - & \mu\pi_T \\ \mu\pi_A & \mu\pi_C & \mu\pi_G & - \end{pmatrix}$$

4. Hasegawa, Kishino and Yano 1985 model: model='HKY85', rate=[μ, κ, π_A, π_C, π_G]

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

$$R = \begin{pmatrix} - & \mu\pi_C & \mu\kappa\pi_G & \mu\pi_T \\ \mu\pi_A & - & \mu\pi_G & \mu\kappa\pi_T \\ \mu\kappa\pi_A & \mu\pi_C & - & \mu\pi_T \\ \mu\pi_A & \mu\kappa\pi_C & \mu\pi_G & - \end{pmatrix}$$

5. Tamura 1992 model: model='T92', rate=[μ, π_{GC}]

   This model is a HKY85 model with $\pi_G = \pi_C = \pi_{GC}/2$ and $\pi_A = \pi_T = \pi_{AT}/2 = (1 - \pi_{GC})/2$,

$$R = \begin{pmatrix} - & \frac{1}{2}\mu\pi_{GC} & \frac{1}{2}\mu\nu\pi_{GC} & \frac{1}{2}\mu\pi_{AT} \\ \frac{1}{2}\mu\pi_{AT} & - & \frac{1}{2}\mu\pi_{GC} & \frac{1}{2}\mu\nu\pi_{AT} \\ \frac{1}{2}\mu\nu\pi_{AT} & \frac{1}{2}\mu\pi_{GC} & - & \frac{1}{2}\mu\pi_{AT} \\ \frac{1}{2}\mu\pi_{AT} & \frac{1}{2}\mu\nu\pi_{GC} & \frac{1}{2}\mu\pi_{GC} & - \end{pmatrix}$$

6. Tamura and Nei 1993 model: model='TN93', rate=[μ, κ_1, κ_2, π_A, π_C, π_G]

   This model extends the HKY1985 model by distinguishing $A \longleftrightarrow G$ transitions (namely $0 \longleftrightarrow 2$) and $C \leftrightarrow T$ transitions ($1 \longleftrightarrow 3$) with different κ.

$$R = \begin{pmatrix} - & \mu\pi_C & \mu\kappa_1\pi_G & \mu\pi_T \\ \mu\pi_A & - & \mu\pi_G & \mu\kappa_2\pi_T \\ \mu\kappa_1\pi_A & \mu\pi_C & - & \mu\pi_T \\ \mu\pi_A & \mu\kappa_2\pi_C & \mu\pi_G & - \end{pmatrix}$$

7. Generalized time reversible model: model='GTR', rate=[x_1, x_2, x_3, x_4, x_5, x_6, π_A, π_C, π_G]

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

$$R = \begin{pmatrix} - & \frac{\pi_A x_1}{\pi_C} & \frac{\pi_A x_2}{\pi_G} & \frac{\pi_A x_3}{\pi_T} \\ x_1 & - & \frac{\pi_C x_4}{\pi_G} & \frac{\pi_C x_5}{\pi_T} \\ x_2 & x_4 & - & \frac{\pi_G x_6}{\pi_T} \\ x_3 & x_5 & x_6 & - \end{pmatrix}$$

8. General model: model='general' (default), rate=[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}].

   This is the most general model with 12 parameters:

$$R = \begin{pmatrix} - & x_1 & x_2 & x_3 \\ x_4 & - & x_5 & x_6 \\ x_7 & x_8 & - & x_9 \\ x_{10} & x_{11} & x_{12} & - \end{pmatrix}$$

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

Example 3.25 applies a Kimmura's 2-parameter mutation model to a population with a single nucleotide marker.

Example 3.25: A Kimura's 2 parameter mutation model

```
>>> pop = population(size=[2000], loci=1,
...     alleleNames=['A', 'C', 'G', 'T'])
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = [initByFreq([.1, .1, .1, .7])],
...     ops = [
...         acgtMutator(rate=[1e-4, 0.5], model='K80'),
...         stat(alleleFreq=0, step=100),
...         pyEval(r"', '.join(['%.3f' % alleleFreq[0][x] for x in range(4)]) + '\n'",
...             step=100),
...     ],
...     gen=500
... )
0.097, 0.102, 0.099, 0.702
0.085, 0.096, 0.133, 0.687
0.028, 0.157, 0.190, 0.626
0.124, 0.132, 0.149, 0.596
0.093, 0.145, 0.149, 0.613
(500,)
>>>
```

3.5.5 Mutation model for microsatellite markers (`smmMutator`)

The **stepwise mutation model** (SMM) was proposed by Ohta and Kimura [1973] to model the mutation of Variable Number Tandem Repeat (VNTR), which consists of tandem repeat of sequences. VNTR markers consisting of short sequences (e.g. 5 basepair or less) are also called microsatellite markers. A mutation event of a VNTR marker either increase of decrease the number of repeats, as a result of slipped-strand mispairing or unequal sister chromatid exchange and genetic recombination.

A `smmMutator` assumes that alleles at a locus are the number of tandem repeats and mutates them by increasing or decreasing the number of repeats during a mutation event. By adjusting parameters `incProb`, `maxAllele` and `mutStep`, this operator can be used to simulate the standard neutral stepwise mutation model and a number of **generalized stepwise mutation models**. For example, Example 3.26 uses two `smmMutator` to mutate two microsatellite

markers, using a standard and a generalized model where a geometric distribution is used to determine the number of steps.

Example 3.26: A standard and a generalized stepwise mutation model

```
>>> simu = simulator(population(size=1000, loci=[1, 1]), randomMating())
>>> simu.evolve(
...     # all start from allele 50
...     preOps = [initByFreq( [0]*50 + [1])],
...     ops = [
...         smmMutator(rates=1e-3, loci=0),
...         smmMutator(rates=1e-3, incProb=0.6, loci=1,
...             mutStep=(GeometricDistribution, 0.2)),
...     ],
...     gen=100
... )
(100,)
>>> # count the average number tandem repeats at both loci
>>> cnt0 = cnt1 = 0
>>> for ind in simu.population(0).individuals():
...     cnt0 += ind.allele(0, 0) + ind.allele(0, 1)
...     cnt1 += ind.allele(1, 0) + ind.allele(1, 1)
...
>>> print 'Average number of repeats at two loci are %.2f and %.2f.' % \
...     (cnt0/2000., cnt1/2000.)
Average number of repeats at two loci are 49.96 and 49.82.
>>>
>>>
```

3.5.6 Simulating arbitrary mutation models using a hybrid mutator (`pyMutator`)*

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

Example 3.27 applies a simple mutation model where an allele is increased by a random number between 1 and 5 when it is mutated. Two different mutation rates are used for two different loci so average alleles at these two loci are different.

Example 3.27: A hybrid mutation model

```
>>> import random
>>> def incAllele(allele):
...     return allele + random.randint(1, 5)
...
>>> simu = simulator(population(size=1000, loci=[20]),
...     randomMating())
>>> simu.evolve(
...     preOps = initSex(),
...     ops = [
...         pyMutator(func=incAllele, rates=[1e-4, 1e-3],
...             loci=[2, 10])
...     ],
...     gen = 1000
... )
(1000,)
>>> # count the average number tandem repeats at both loci
>>> def avgAllele(pop, loc):
```

```
...     ret = 0
...     for ind in pop.individuals():
...         ret += ind.allele(loc, 0) + ind.allele(loc, 1)
...     return ret / (pop.popSize() * 2.)
...
>>> pop = simu.population(0)
>>> print 'Average number of repeats at two loci are %.2f and %.2f.' % \
...     (avgAllele(pop, 2), avgAllele(pop, 10))
Average number of repeats at two loci are 0.03 and 1.71.
>>>
```

3.5.7 Mixed mutation models (`mixedMutator`) **

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

A `mixedMutator` is constructed by a list of mutators and their respective probabilities. It accepts regular mutator parameters such as `rates`, `loci`, `subPops`, `mapIn and mapOut` and mutates aleles at specified rate. When a mutation event happens, it calls one of the mutators to mutate the allele. For example, Example 3.28 applies a mixture of k-allele model and stepwise model to mutate a micosatellite model.

Example 3.28: A mixed k-allele and stepwise mutation model

```
>>> simu = simulator(population(5000, loci=[1, 1]),
...     randomMating())
>>> simu.evolve(
...     preOps = initByValue([50, 50]),
...     ops = [
...         # the first locus uses a pure stepwise mutation model
...         smmMutator(rates=0.001, loci=0),
...         # the second locus uses a mixed model
...         mixedMutator(rates=0.001, loci=1, mutators=[
...             kamMutator(rates=1, k=100),
...             smmMutator(rates=1)
...         ], prob=[0.1, 0.9])],
...     gen = 20
... )
(20,)
>>> # what alleles are there?
>>> geno0 = []
>>> geno1 = []
>>> for ind in simu.population(0).individuals():
...     geno0.extend([ind.allele(0, 0), ind.allele(0, 1)])
...     geno1.extend([ind.allele(1, 0), ind.allele(1, 1)])
...
>>> print 'Locus 0 has alleles', ', '.join([str(x) for x in set(geno0)])
Locus 0 has alleles 49, 50, 51
>>> print 'Locus 1 has alleles', ', '.join([str(x) for x in set(geno1)])
Locus 1 has alleles 49, 50, 51, 52, 54, 30
>>>
```

When a mutation event happens, mutators in Example 3.28 mutate the allele with probability (mutation rate) 1. If different mutation rates are specified, the overall mutation rates would be the product of mutation rate of `mixedMutator` and the passed mutators. However, it is extremely important to understand that although `mixedMutator(rates=mu)` with `smmMutator(rates=1)` and `mixedMutator(rates=1)` with

`smmMutator(rates=mu)` mutate alleles at the same mutation rate, the former is much more efficient because it triggers far less mutation events.

3.5.8 Context-dependent mutation models (`contextMutator`)**

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

A `contextMutator` can be used to mutate an allele depending on its surrounding loci. This mutator is constructed by a list of mutators and their respective contexts. It accepts regular mutator parameters such as `rates`, `loci`, `subPops`, `mapIn and mapOut` and mutates aleles at specified rate. When a mutation event happens, it checks the context of the mutaed allele and choose a corresponding mutator to mutate the allele. An additional mutator can be specified to mutate alleles with unknown context. Example 3.29 applies two `snpMutator` at different rates under different contexts.

Example 3.29: A context-dependent mutation model

```
>>> simu = simulator(population(5000, loci=[3, 3]),
...     randomMating())
>>> simu.evolve(
...     # initialize locus by 0, 0, 0, 1, 0, 1
...     preOps = initByValue([1, 1], loci=[3, 5]),
...     ops = [
...         contextMutator(mutators=[
...             snpMutator(u=0.1),
...             snpMutator(u=1),
...             ],
...             contexts=[(0, 0), (1, 1)],
...             loci=[1, 4],
...             rates=0.01
...         ),
...         stat(alleleFreq=[1, 4], step=5),
...         pyEval(r"'Gen: %2d freq1: %.3f, freq2: %.3f\n'" +
...             " % (gen, alleleFreq[1][1], alleleFreq[4][1])", step=5)
...     ],
...     gen = 20
... )
Gen:  0 freq1: 0.001, freq2: 0.009
Gen:  5 freq1: 0.006, freq2: 0.058
Gen: 10 freq1: 0.009, freq2: 0.110
Gen: 15 freq1: 0.015, freq2: 0.153
(20,)
>>>
```

Note that although

```
contextMutator(mutators=[
    snpMutator(u=0.1),
    snpMutator(u=1)],
    contexts=[(0, 0), (1, 1)],
    rates=0.01
)
```

and

```
contextMutator(mutators=[
```

```
    snpMutator(u=0.001),
    snpMutator(u=0.01)],
    contexts=[(0, 0), (1, 1)],
    rates=1
)
```

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

Context-dependent mutator can also be implemented by a `pyMutator`. When a non-zero parameter `context` is specified, this mutator will collect `context` number of alleles to the left and right of a mutated allele and pass them as a second parameter of the user-provided mutation function. Example 3.30 applies the same mutation model as Example 3.29 using a `pyMutator`.

Example 3.30: A hybrid context-dependent mutation model

```
>>> import random
>>>
>>> simu = simulator(population(5000, loci=[3, 3]),
...     randomMating())
>>>
>>> def contextMut(allele, context):
...     if context == [0, 0]:
...         if allele == 0 and random.random() < 0.1:
...             return 1
...     elif context == [1, 1]:
...         if allele == 0:
...             return 1
...     # do not mutate
...     return allele
...
>>> simu.evolve(
...     # initialize locus by 0, 0, 0, 1, 0, 1
...     preOps = initByValue([1, 1], loci=[3, 5]),
...     ops = [
...         pyMutator(func=contextMut, context=1,
...             loci=[1, 4],  rates=0.01
...         ),
...         #snpMutator(u=0.01, v= 0.01, loci=[1, 4]),
...         stat(alleleFreq=[1, 4], step=5),
...         pyEval(r"'Gen: %2d freq1: %.3f, freq2: %.3f\n'" +
...             " % (gen, alleleFreq[1][1], alleleFreq[4][1])", step=5)
...     ],
...     gen = 20
... )
Gen:  0 freq1: 0.001, freq2: 0.010
Gen:  5 freq1: 0.005, freq2: 0.057
Gen: 10 freq1: 0.010, freq2: 0.108
Gen: 15 freq1: 0.015, freq2: 0.154
(20,)
>>>
```

3.5.9 Manually-introduced mutations (`pointMutator`)

Operator `pointMutator` is different from all other mutators in that it mutates specified alleles of specified individuals. It is usually used to manually introduce one or more mutants to a population. Although it is not a recommended method to introduce a disease predisposing allele, the following example (Example 3.31) demonstrates an evolutionary process where mutants are repeatedly introduced and raised by positive selection until it reaches an appreciable allele

frequency. This example uses two `ifElse` operators. The first one introduces a mutant when there is no mutant in the population, and the second one terminate the evolution when the frequency of the mutant reaches 0.05.

Example 3.31: Use a point mutator to introduce a disease predisposing allele

```
>>> pop = population(1000, loci=1, infoFields='fitness')
>>> simu = simulator(pop, randomSelection())
>>> simu.evolve(
...     preOps = pyOutput('Introducing alleles at generation'),
...     ops = [
...         stat(alleleFreq=0),
...         ifElse('alleleNum[0][1] == 0', ifOps=[
...             pyEval(r"' %d' % gen"),
...             pointMutator(inds=0, loci=0, allele=1),
...         ]),
...         maSelector(loci=0, wildtype=0, fitness=[1, 1.05, 1.1]),
...         ifElse('alleleFreq[0][1] > 0.05', ifOps=[
...             pyEval(r"'.\nTerminate at generation %d at allele freq %.3f.\n'" +
...                 " % (gen, alleleFreq[0][1])"),
...             terminateIf('True'),
...         ])
...     ],
... )
Introducing alleles at generation 0 10 11 17 79.
Terminate at generation 119 at allele freq 0.051.
(120,)
>>>
```

3.5.10 Apply mutation to (virtual) subpopulations *

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

Virtual subpopulations can also be specified in this parameter. For example, you can apply different mutation models to male and female individuals, to unaffected or affected individuals, to patients at different stages of a cancer. Example 3.32 demonstrate a mutation model where individuals with more tandem repeats at a disease predisposing locus are more likely to develop a disease (e.g. fragile-X). Affected individuals are then subject to a non-neutral mutation model at an accerlerated mutation rate.

Example 3.32: Applying mutation to virtual subpopulations.

```
>>> def fragileX(geno):
...     '''A disease model where an individual has increased risk of
...     affected if the number of tandem repeats exceed 75.
...     '''
...     # Alleles A1, A2.
...     maxRep = max(geno)
...     if maxRep < 50:
...         return 0
...     else:
...         # individuals with allele >= 70 will surely be affected
...         return min(1, (maxRep - 50)*0.05)
...
>>> def avgAllele(pop):
...     'Get average allele by affection status.'
...     Stat(pop, alleleFreq=(0,1), subPops=[(0,0), (0,1)],
...         numOfAffected=True, vars=['alleleNum', 'alleleNum_sp'])
...     avg = []
```

```
...     for alleleNum in [\
...             pop.dvars((0,0)).alleleNum[0],  # first locus, unaffected
...             pop.dvars((0,1)).alleleNum[0],  # first locus, affected
...             pop.dvars().alleleNum[1],       # second locus, overall
...         ]:
...         alleleSum = numAllele = 0
...         for idx,cnt in enumerate(alleleNum):
...             alleleSum += idx * cnt
...             numAllele += cnt
...         if numAllele == 0:
...             avg.append(0)
...         else:
...             avg.append(alleleSum * 1.0 /numAllele)
...     # unaffected, affected, loc2
...     pop.dvars().avgAllele = avg
...     return True
...
>>> pop = population(10000, loci=[1, 1])
>>> pop.setVirtualSplitter(affectionSplitter())
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = initByValue([50, 50]),
...     ops = [
...         # determine affection status for each offspring (duringMating)
...         pyPenetrance(func=fragileX, loci=0),
...         # unaffected offspring, mutation rate is high to save some time
...         smmMutator(rates=1e-3, loci=1),
...         # unaffected offspring, mutation rate is high to save some time
...         smmMutator(rates=1e-3, loci=0, subPops=[(0, 0)]),
...         # affected offspring have high probability of mutating upward
...         smmMutator(rates=1e-2, loci=0, subPops=[(0, 1)],
...            incProb=0.7, mutStep=3),
...         # number of affected
...         pyOperator(func=avgAllele, step=20),
...         pyEval(r"'Gen: %3d #Aff: %d AvgRepeat: %.2f (unaff), %.2f (aff), %.2f (unrelated)\n'"
...             + " % (gen, numOfAffected, avgAllele[0], avgAllele[1], avgAllele[2])",
...             step=20),
...     ],
...     gen = 101
... )
Gen:   0 #Aff: 0 AvgRepeat: 1.01 (unaff), 0.00 (aff), 1.01 (unrelated)
Gen:  20 #Aff: 12 AvgRepeat: 1.53 (unaff), 0.50 (aff), 1.53 (unrelated)
Gen:  40 #Aff: 19 AvgRepeat: 2.56 (unaff), 1.52 (aff), 1.52 (unrelated)
Gen:  60 #Aff: 25 AvgRepeat: 3.08 (unaff), 1.52 (aff), 2.04 (unrelated)
Gen:  80 #Aff: 51 AvgRepeat: 3.09 (unaff), 2.00 (aff), 2.04 (unrelated)
Gen: 100 #Aff: 48 AvgRepeat: 3.09 (unaff), 1.54 (aff), 2.56 (unrelated)
(101,)
>>>
>>>
```

At the beginning of a simulation, all individuals have 50 copies of a tandem repeat and the mutation follows a standard neutral stepwise mutation model. Individuals with more than 50 repeats will have an increasing probability to develop a disease ($\Pr(\text{affected} \mid n) = (n - 50) * 0.05$) for $50 \leq n \leq 70$). The averge repeat number therefore increases for affected individuals. In contrast, the mean number of repeats at locus 1 on a separate chromosome oscillate around 50.

3.5.11 Allele mapping **

If alleles in your simulation do not follow the convention of a mutation model, you will have to use a general mutation model with your own mutation matrix, or an advanced feature called **allele mapping**.

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

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

to map your alleles (`C(0)->C(1), G(1)->G(2), A(2)->A(0), T(3)->T(3)`) to alleles `acgtMutator` assumes, and then map mutated alleles (`A(0)->A(2), C(1)->C(0), G(2)->G(1), T(3)->T(3)`) back. Example 3.33 gives another example where alleles 4, 5, 6 and 7 are mutated using a 4-allele model.

Example 3.33: Allele mapping for mutation operators

```
>>> pop = population(size=[2000], loci=1)
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = initByFreq([0]*4 + [0.1, 0.2, 0.3, 0.4]),
...     ops = [
...         kamMutator(k=4, rates=1e-4, mapIn=[0]*4 + range(4),
...             mapOut=[4, 5, 6, 7]),
...         stat(alleleFreq=0, step=100),
...         pyEval(r"', '.join(['%.2f' % alleleFreq[0][x] for x in range(8)]) + '\n'",
...             step=100),
...     ],
...     gen=500
... )
0.00, 0.00, 0.00, 0.00, 0.10, 0.20, 0.31, 0.40
0.00, 0.00, 0.00, 0.00, 0.04, 0.23, 0.45, 0.28
0.00, 0.00, 0.00, 0.00, 0.02, 0.36, 0.31, 0.31
0.00, 0.00, 0.00, 0.00, 0.00, 0.33, 0.32, 0.36
0.00, 0.00, 0.00, 0.00, 0.00, 0.31, 0.21, 0.48
(500,)
>>>
```

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

3.5.12 Infinite-sites model and mutation-introduced loci **

All mutation models in simuPOP apply to existing alleles at pre-specified loci. However, if the location of loci cannot be determined beforehand, it is sometimes desired to create new loci as a result of mutation. A customized operator can be used for this purpose (see Example 2.36), but extra attention is needed to make sure that other operators are applied to the correct loci because loci indexes will be changed with the insertion of new loci.

Infinite-sites and infinite-alleles models have some similarities. In some cases, you can treat a long chromosomal region as a locus and use the infinite-alleles model, actually a k-allele model with large k, to mimic the infinite-site model. Because there is supposed to be only one mutant at each site, you can assign a unique *location* for each allele of an infinite-allele model and convert multi-allelic datasets simulated by an infinite-allele model to sequences of diallelic markers. It worth noting that mutation rates are interpreted differently for these two models.

If you really would like to simulate mutations over long sequences (e.g., mutate each basepair at a multi-centiMorgan

region), simulating millions of markers might not be realistic. Allowing mutations to insert new loci can be more efficient, although this method has its own limitations.

3.5.13 Mutation rate and transition matrix of a `matrixMutator` **

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

Because alleles usually have different probabilities of mutating to other alleles, **a mutation process is usually allele dependent**. Given a mutation model (p_{ij}), it is obviously inefficient to go through all mutable alleles and determine whether or not to mutate it using $p_{ij}, j = 0, ..., 1-n$. simuPOP uses a two step procedure to mutate a large number of alleles. More specifically, for each mutation model, we determine $\mu = \max_{i=0}^{n-1}(1-p_{ii})$ as the overall mutation rate, and then

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

$$\Pr(i \to j) = \begin{cases} 1 - \frac{1}{\mu}(1 - p_{ii}) & \text{if } i = j \\ \frac{p_{ij}}{\mu} & \text{if } i \neq j \end{cases}$$

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

$$p_{ij} = \mu \Pr(i \to j)$$

if $i \neq j$ and

$$p_{ii} = (1 - \mu) + \mu \Pr(i \to i)$$

where the first and second items are probabilities of no-mutation at steps 1 and 2. μ was chosen as the smallest μ that makes $0 \leq \Pr(i \to i) \leq 1$ for all i.

For example, for a k-allele model with

$$p_{ij} = \begin{pmatrix} 1-\mu & \frac{\mu}{k-1} & \cdots & \frac{\mu}{k-1} \\ \frac{\mu}{k-1} & 1-\mu & \cdots & \frac{\mu}{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mu}{k-1} & \frac{\mu}{k-1} & \cdots & 1-\mu \end{pmatrix}$$

μ is directly μ for the first step and

$$\Pr(i \to j) = \begin{cases} 0 & \text{if } i = j \\ \frac{1}{k-1} & \text{if } i \neq j \end{cases}$$

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

For a classical mutation model with $P(A \to a) = u$ and $P(a \to A) = v$,

$$p_{ij} = \begin{pmatrix} 1-u & u \\ v & 1-v \end{pmatrix}$$

if $u = 0.001$ and $v = 0.0005$, $\mu = \max(u, v) = 0.001$,

$$\Pr(i \to j) = \begin{pmatrix} 0 & 1 \\ \frac{v}{u} = 0.5 & 1 - \frac{v}{u} = 0.5 \end{pmatrix}$$

That is to say, we would mutate at a mutation rate $u = 0.001$, mutate allele A to a with probability 1 and mutate allele a to A with probability 0.5.

3.6 Statistics calculation (operator `stat`)

3.6.1 How statistics calculation works

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

Statistics calculated by the `stat` operator are stored in a population's local namespace, which can be retrieved using functions `population.vars()` or `population.dvars()`. The only difference between these functions is that `vars()` returns a dictionary and `dvars()` returns a Python object that uses variable names as attributes (`vars()['alleleFreq']` is equivalent to `dvars().alleleFreq`). More importantly, arbitrary expressions and statements can be evalulated in a population's local namespace which allows operators such as `pyEval` and `pyExec` to manipuate and output statistics in any format.

3.6.2 Support for virtual subpopulations

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

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

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

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

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

tells this operator to output numbers and frequencies of alleles at locus `0` in subpopulation `0` and virtual subpopulation `(1,0)`. These variables will be saved in dictionaries `subPop[sp]` of the local namespace. For example, the above operator will write variables such as `subPop[0]['alleleFreq']`, `subPop[(1,0)]['alleleFreq'] and subPop[(1,0)]['alleleNum']`. Functions `population.vars(sp)` and `population.dvars(sp)` are provided as shortcuts to access these variables but the full variable names have to be specified if these variables are used in expressions. **Note:** The `stat` opeartor accepts overlapping or even duplicate virtual subpopulations. During the calculation of summary statistics, these subpopulations are treated as separate subpopulations so some individuals can be counted more than once. For example, individuals in virtual subpopulation (0, 1) will be counted twice during the calculation of allele frequency and population size in operator

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

3.6.3 Counting individuals by sex and affection status

Parameters *popSize*, *numOfMale* and *numOfAffected* provide basic individual counting statistics. They count the number of all, male/female, affected/unaffected individuals in all or specified (virtual) subpopulations, and set variables such as `popSize`, `numOfMale`, `numOfFemale`, `numOfAffected`, `numOfUnaffected`. Proportions and statistics for subpopulations are available if variables such as `propOfMale`, `numOfAffected_sp` are specified in

parameter vars. Another variable `subPopSize` is defined for parameter `popSize=True`. It is a list of sizes of all or specified subpopulations and is easier to use than referring to variable `popSize` from individual subpopulations.

Example 3.34 demonstrates how to use these parameters in operator `stat`. It defines four VSPs by sex and affection status (using a `combinedSplitter`) and count individuals by sex and affection status. It is worth noting that `pop.dvars().popSize` in the first example is the total number of individuals in two virtual subpopulations `(0,0)` and `(0,2)`, which are all male indiviudals, and all unaffected individuals. Because these two VSPs overlap, this variable can be larger than actual population size.

Example 3.34: Count individuals by sex and/or affection status

```
>>> pop = population(10000, loci=1)
>>> pop.setVirtualSplitter(combinedSplitter(
...     [sexSplitter(), affectionSplitter()]))
>>> InitByFreq(pop, [0.2, 0.8])
>>> MaPenetrance(pop, loci=0, penetrance=[0.1, 0.2, 0.5])
>>> # Count population size
>>> Stat(pop, popSize=True, subPops=[(0, 0), (0, 2)])
>>> # popSize is the size of two VSPs, does not equal to total population size.
>>> # Because two VSPs overlap (all males and all unaffected), popSize can be
>>> # greater than real population size.
>>> print pop.dvars().subPopSize, pop.dvars().popSize
[4930, 6073] 11003
>>> # print popSize of each virtual subpopulation.
>>> Stat(pop, popSize=True, subPops=[(0, 0), (0, 2)], vars='popSize_sp')
>>> # Note the two ways to access variable in (virtual) subpopulations.
>>> print pop.dvars((0,0)).popSize, pop.dvars().subPop[(0,2)]['popSize']
4930 6073
>>> # Count number of male (should be the same as the size of VSP (0,0).
>>> Stat(pop, numOfMale=True)
>>> print pop.dvars().numOfMale
4930
>>> # Count the number of affected and unaffected male individual
>>> Stat(pop, numOfMale=True, subPops=[(0, 2), (0, 3)], vars='numOfMale_sp')
>>> print pop.dvars((0,2)).numOfMale, pop.dvars((0,3)).numOfMale
2964 1966
>>> # or number of affected male and females
>>> Stat(pop, numOfAffected=True, subPops=[(0, 0), (0, 1)], vars='numOfAffected_sp')
>>> print pop.dvars((0,0)).numOfAffected, pop.dvars((0,1)).numOfAffected
1966 1961
>>>
```

3.6.4 Allele count and frequency

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

Example 3.35 demonstrates an advanced usage of allele counting statistic. In this example, two virtual subpopulations are defined by individual affection status. During evolution, a multi-allele penetrance operator is used to determine individual affection status and a `stat` operator is used to calculate allele frequencies in these two virtual subpopulations, and in the whole population. Because the simulated disease is largely caused by the existence of allele 1 at the first locus, it is expected that the frequency of allele 1 is higher in the case group than in the control group. It is worth

noting that `alleleFreq[0][1]` in this example is the frequency of allele 1 in the whole population because these two virtual subpopulations add up to the whole population.

Example 3.35: Calculate allele frequency in affected and unaffected individuals

```
>>> pop = population(10000, loci=1)
>>> pop.setVirtualSplitter(affectionSplitter())
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = [initByFreq(loci=0, alleleFreq=[0.8, 0.2])],
...     ops = [
...         maPenetrance(penetrance=[0.1, 0.4, 0.6], loci=0),
...         stat(alleleFreq=0, subPops=[(0, 0), (0, 1)],
...             vars=['alleleFreq', 'alleleFreq_sp']),
...         pyEval(r"'Gen: %d, freq: %.2f, freq (aff): %.2f, freq (unaff): %.2f\n' % " + \
...             "(gen, alleleFreq[0][1], subPop[(0,1)]['alleleFreq'][0][1]," + \
...             "subPop[(0,0)]['alleleFreq'][0][1])"),
...     ],
...     gen = 5
... )
Gen: 0, freq: 0.20, freq (aff): 0.40, freq (unaff): 0.14
Gen: 1, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 2, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.14
Gen: 3, freq: 0.20, freq (aff): 0.41, freq (unaff): 0.15
Gen: 4, freq: 0.20, freq (aff): 0.40, freq (unaff): 0.15
(5,)
>>>
```

3.6.5 Genotype count and frequency

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

Example 3.36 creates a small population and initializes a locus with rare alleles 0, 1 and a common allele 2. A function `Stat` (the function form of operator `stat`) is used to count the available genotypes. Note that `pop.dvars().genoFreq[0][(i,j)]` can be used to print frequencies of all genotypes even when not all genotypes are available in the population.

Example 3.36: Counting genotypes in a population

```
>>> pop = population(100, loci=[1, 1, 1], chromTypes=[Autosome, ChromosomeX, ChromosomeY])
>>> InitByFreq(pop, [0.01, 0.05, 0.94])
>>> Stat(pop, genoFreq=[0, 1])
>>> print 'Available genotypes on autosome:', pop.dvars().genoFreq[0].keys()
Available genotypes on autosome: [(1, 2), (2, 0), (2, 1), (2, 2)]
>>> for i in range(3):
...     for j in range(3):
...         print '%d-%d: %.3f' % (i, j, pop.dvars().genoFreq[0][(i,j)])
...
0-0: 0.000
0-1: 0.000
0-2: 0.000
1-0: 0.000
1-1: 0.000
```

```
1-2: 0.030
2-0: 0.020
2-1: 0.030
2-2: 0.920
>>> print 'Genotype frequency on chromosome X:\n', \
...     '\n'.join(['%s: %.3f' % (x,y) for x,y in pop.dvars().genoFreq[1].iteritems()])
Genotype frequency on chromosome X:
(1, 2): 0.030
(1,): 0.050
(2,): 0.500
(2, 1): 0.020
(2, 0): 0.010
(2, 2): 0.390
>>>
>>>
```

3.6.6 Homozygote and heterozygote count and frequency

In a diploid population, a heterozygote is a genotype with two different alleles and a homozygote is a genotype with two identical alleles. Parameter `heteroFreq` accepts a list of loci and outputs variables `heteroFreq` which is a dictionary of heterozygote frequencies at specfied loci. Optional variables `heteroNum`, `homoFreq` and `homoNum` can be outputted for all and each (virtual) subpopulations. Example 3.37 demonstrates the decay of heterozygosity of a locus due to genetic drift.

Example 3.37: Counting homozygotes and heterozygotes in a population

```
>>> pop = population(100, loci=1)
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = initByFreq([0.5, 0.5]),
...     ops = [
...         stat(heteroFreq=0, step=10),
...         pyEval(r"'Gen: %d, HeteroFreq: %.2f\n' % (gen, heteroFreq[0])", step=20)
...     ],
...     gen = 100
... )
Gen: 0, HeteroFreq: 0.51
Gen: 20, HeteroFreq: 0.48
Gen: 40, HeteroFreq: 0.51
Gen: 60, HeteroFreq: 0.53
Gen: 80, HeteroFreq: 0.45
(100,)
>>>
```

3.6.7 Haplotype count and frequency

Haplotypes refer to alleles on the same homologous copy of a chromosome at specified loci. For example, an diploid individual can have haplotypes `(0, 2, 1)` and `(0, 1, 1)` at loci `(2, 3, 5)` if he or she has genotype `(0, 0)`, `(2, 1)` and `(1,1)` at loci 2, 3 and 5 respectively. Parameter *haploFreq* accept one or more lists of loci specifying one or more haplotype sites (e.g. `haploFreq=[(0,1,2), (2,3)]` specifies two haplotype sites). The results are saved to dictionaries (with haplotype site as keys) of default dictionaries (with haplotype as keys). For example, `haploFreq[(0,1,2)][(0,1,1)]` will be the frequency of haplotype `(0, 1, 1)` at loci `(0, 1, 2)`. Example 3.38 prints the numbers of genotypes and haplotypes at loci 0, 1 and 2 of a small population. Note that the `ViewVars` function defined in module `simuUtil` can make use of a wxPython window to view all variables if it is called in GUI mode.

Example 3.38: Counting haplotypes in a population

```
>>> pop = population(100, loci=3)
>>> InitByFreq(pop, [0.2, 0.4, 0.4], loci=0)
>>> InitByFreq(pop, [0.2, 0.8], loci=2)
>>> Stat(pop, genoFreq=[0, 1, 2], haploFreq=[0, 1, 2],
...     vars=['genoNum', 'haploFreq'])
>>> from simuUtil import ViewVars
>>> ViewVars(pop.vars(), gui=False)
{'genoNum': {0: {(0, 0): 5.0,
                 (0, 1): 6.0,
                 (0, 2): 9.0,
                 (1, 0): 7.0,
                 (1, 1): 17.0,
                 (1, 2): 13.0,
                 (2, 0): 5.0,
                 (2, 1): 20.0,
                 (2, 2): 18.0},
             1: {(0, 0): 100.0},
             2: {(0, 1): 16.0, (1, 0): 17.0, (0, 0): 3.0, (1, 1): 64.0}},
 'haploFreq': {(0, 1, 2): {(0, 0, 0): 0.02,
                           (0, 0, 1): 0.16500000000000001,
                           (1, 0, 0): 0.074999999999999997,
                           (1, 0, 1): 0.32500000000000001,
                           (2, 0, 0): 0.10000000000000001,
                           (2, 0, 1): 0.315}}}
>>>
```

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

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

3.6.8 Summary statistics of information fields

Parameter `sumOfInfo`, `meanOfInfo`, `varOfInfo`, `maxOfInfo` and `minOfInfo` are used to calculate the sum, mean, sample variance ($\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2$), max and min of specified information fields of individuals in all or specified (virtual) subpopulations. The results are saved in dictionaries `sumOfInfo`, `meanOfInfo`, `varOfInfo`, `maxOfInfo` and `minOfInfo` with information fields as keys. For example, parameter `meanOfInfo='age'` calculates the mean age of all individuals and set variable `meanOfInfo['age']`.

Example 3.39 demonstrates a mixing process of two populations. The population starts with two types of individuals with ancestry values 0 or 1 (information field `anc`). During the evolution, parents mate randomly and the ancestry of offspring is the mean of parental ancestry values. A `stat` operator is used to calculate the mean and variance of individual ancestry values, and the number of individuals in five ancestry groups. It is not surprising that whereas population mean ancestry does not change, more and more people have about the same number of ancestors from each group and have an ancestry value around 0.5. The variance of ancestry values therefore decreases gradually.

Example 3.39: Calculate summary statistics of information fields

```
>>> import random
>>> pop = population([500], infoFields='anc')
>>> # anc is 0 or 1
>>> pop.setIndInfo([random.randint(0, 1) for i in range(500)], 'anc')
>>> # Defines VSP 0, 1, 2, 3, 4 by anc.
>>> pop.setVirtualSplitter(infoSplitter('anc', cutoff=[0.2, 0.4, 0.6, 0.8]))
>>> #
>>> def passInfo(fields):
```

```
...     'Parental fields will be passed as anc1, anc2'
...     return (fields[0] + fields[1]) / 2.
...
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     preOps = initSex(),
...     ops = [
...         pyTagger(passInfo, infoFields='anc'),
...         stat(popSize=True, meanOfInfo='anc', varOfInfo='anc',
...             subPops=[(0,x) for x in range(5)]),
...         pyEval(r"'Anc: %.2f (%.2f), #inds: %s\n' %" + \
...             "(meanOfInfo['anc'], varOfInfo['anc'], " + \
...             "', '.join(['%4d' % x for x in subPopSize]))")
...     ],
...     gen = 5,
... )
Anc: 0.53 (0.13), #inds:  116,    0,  237,    0,  147
Anc: 0.54 (0.07), #inds:   28,  110,  167,  149,   46
Anc: 0.54 (0.03), #inds:   16,  124,  137,  186,   37
Anc: 0.54 (0.02), #inds:    2,   73,  249,  165,   11
Anc: 0.54 (0.01), #inds:    0,   28,  352,  120,    0
(5,)
>>>
>>>
```

3.6.9 Linkage disequilibrium

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

Assuming that two loci are both diallelic, one with alleles A and a, and the other with alleles B and b. If we denote P_x, P_{xy} as allele and haplotype frequencies for allele x and haplotype xy, respectively, the linkage disequilibrium measures **with respect to primaries alleles** *A* and *B* are

- Basic LD measure D:

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

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

- Lewontin's $D' = D/D_{max}$ where

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

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

- r^2 (Δ^2 in Devlin and Risch [1995])

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

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

- If primary alleles are specified, all other alleles are considered as minor alleles with combined frequency (e.g. $1 - P_A$). The same formulas apply which lead to signed D and D' measures.

- If primary alleles are not specified, these LD measures are calculated as the average of the absolute value of diallelic measures of all allele pairs. For example, the multi-allele version of r^2 is

$$r^2 = \sum_i \sum_j P_i P_j \left| r_{ij}^2 \right| = \sum_i \sum_j \frac{D_{ij}^2}{(1-P_i)(1-P_j)}$$

  where i and j iterate through all alleles at the two loci. **In the diallelic case, LD measures will be the absolute value of the single measures** because D_{ij} and D'_{ij} only differ by signs.

In another word,

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

Note: A large number of linkage disequilibrium measures have been used in different disciplines but not all of them are well-accepted. Requests of adding a particular LD measure will be considered when a reliable reference is provided. Association tests between specified loci could also be calculated using a m by n table of haplotype frequencies. If primary alleles are specified, non-primary alleles are combined to form a 2 by 2 table ($m = n = 2$). Otherwise, m and n are respective numbers of alleles at two loci.

- χ^2 and its p-value (variable `LD_ChiSq` and `LD_ChiSq_p`, respectively). A one-side χ^2 test with $(m-1) \times (n-1)$ degrees of freedom will be used.
- Cramer V statistic (variable `CramerV`):

$$V = \sqrt{\frac{\chi^2}{N \times \min(m-1, n-1)}}$$

  where N equals the total number of haplotypes ($2 \times$ popSize for autosomes in diploid populations).

This statistic sets variables `LD`, `LD_prime`, `R2`, and optionally `ChiSq`, `ChiSq_p` and `CramerV`. Subpopulation specific variables can be calculated by specifying variables such as `LD_sp` and `R2_sp`. Example 3.40 demonstrates how to calculate various LD measures and output selected variables. Note that the significant overall LD between two loci is an artifact of population structure because loci are in linkage equilibrium in each subpopulation.

Example 3.40: Linkage disequilibrium measures

```
>>> pop = population([1000]*2, loci=3)
>>> InitByFreq(pop, [0.2, 0.8], subPops=0)
>>> InitByFreq(pop, [0.8, 0.2], subPops=1)
>>> Stat(pop, LD=[[0, 1, 0, 0], [1, 2]],
...     vars=['LD', 'LD_prime', 'R2', 'LD_ChiSq', 'LD_ChiSq_p', 'CramerV',
...         'LD_prime_sp', 'LD_ChiSq_p_sp'])
>>>
>>> from pprint import pprint
>>> pprint(pop.vars())
{'CramerV': {0: {1: 0.36464019145063142}, 1: {2: 0.37457939471088936}},
 'LD': {0: {1: 0.091146999999999978}, 1: {2: 0.093638749999999993}},
 'LD_ChiSq': {0: {1: 531.84987688461263}, 1: {2: 561.23889176790499}},
 'LD_ChiSq_p': {0: {1: 0.0}, 1: {2: 0.0}},
 'LD_prime': {0: {1: 0.37256536969592913}, 1: {2: 0.38043075297977252}},
```

```
 'R2': {0: {1: 0.1329624692211531}, 1: {2: 0.14030972294197622}},
 'subPop': {0: {'LD_ChiSq_p': {0: {1: 0.68629558791139345},
                               1: {2: 0.80277847603078512}},
                'LD_prime': {0: {1: 0.0095461505880819336},
                             1: {2: 0.0055935363579863034}}},
            1: {'LD_ChiSq_p': {0: {1: 0.48235543354931376},
                               1: {2: 0.84220734553526855}},
                'LD_prime': {0: {1: 0.015882388421697157},
                             1: {2: 0.018625941883835415}}}}}
>>>
>>>
```

3.6.10 Genetic association

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

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

simuPOP currently supports the following tests:

- **Allele-based Chi-square test**: This is the basic allele-based χ^2 test that can be applied to diploid as well as haploid populations. Basically, a 2 by n contigency table is set up for each locus with n_{ij} being the number of alleles j in cases $(i = 0)$ and controls $(i = 1)$. A χ^2 test is applied to each locus and set variables `Allele_ChiSq` and `Allele_ChiSq_p` to the χ^2 statistic and its two-sided p value (with degrees freedom $n - 1$). Note that genotype information is not preserved in such a test.
- **Genotype-based Chi-square test**: This is the genotype-based χ^2 test for diploid populations. Basically, a 2 by n contigency table is set up for each locus with n_{ij} being the number of genotype j (unordered pairs of alleles) in cases $(i = 0)$ and controls $(i = 1)$. A χ^2 test is applied to each locus and set variables `Geno_ChiSq` and `Geno_ChiSq_p` to the χ^2 statistic and its two-sided p value (with degrees freedom $n - 1$). This test is usually applied to diallelic loci with 3 genotypes (*AA*, *Aa* and *aa*) but it can be applied to loci with more than two alleles as well.
- **Genotype-based trend test**: This Cochran-Armitage test can only be applied to diallelic loci in diploid populations. For each locus, a 2 by 3 contigency table is set up with n_{ij} being the number of genotype j (*AA*, *Aa* and *aa* with *A* being the wildtype allele) in cases $(i = 0)$ and controls $(i = 1)$. A Cochran-Armitage trend test is applied to each locus and set variables `Armitage_p` to its two-sided p value.

Example 3.41 demonstrates how to apply a penetrance model, draw a case-control sample and apply genetic association tests to an evolving population. In this example, a penetrance model is applied to a locus (locus 3). A Python operator is then used to draw a case-control sample from the population and test genetic association at two surrounding loci. Because these two loci are tightly linked to the disease predisposing locus, they are in strong association with the disease initially. However, because of recombination, such association decays with time at rates depending on their genetic distances to the disease predisposing locus.

Example 3.41: Genetic association tests

```
>>> from simuUtil import *
>>> def assoTest(pop):
...     'Draw case-control sample and apply association tests'
...     sample = CaseControlSample(pop, cases=500, controls=500)[0]
...     Stat(sample, association=(0, 2), vars=['Allele_ChiSq_p', 'Geno_ChiSq_p', 'Armitage_p'])
...     print 'Allele test: %.2e, %.2e, Geno test: %.2e, %.2e, Trend test: %.2e, %.2e' \
...         % (sample.dvars().Allele_ChiSq_p[0], sample.dvars().Allele_ChiSq_p[2],
...         sample.dvars().Geno_ChiSq_p[0], sample.dvars().Geno_ChiSq_p[2],
...         sample.dvars().Armitage_p[0], sample.dvars().Armitage_p[2])
...     return True
...
>>> simu = simulator(population(size=100000, loci=3),
...     randomMating())
>>> simu.evolve(
...     preOps = initByValue([[0]*3, [1]*3], proportions=[0.5, 0.5]),
...     ops = [
...         recombinator(loci=[0, 1], rates=[0.01, 0.005]),
...         maPenetrance(loci=1, penetrance=[0.1, 0.2, 0.4]),
...         pyOperator(func=assoTest, step=20),
...     ],
...     gen = 100
... )
Allele test: 0.00e+00, 0.00e+00, Geno test: 0.00e+00, 0.00e+00, Trend test: 0.00e+00, 0.00e+00
Allele test: 0.00e+00, 0.00e+00, Geno test: 1.33e-15, 0.00e+00, Trend test: 1.11e-16, 0.00e+00
Allele test: 5.39e-08, 4.75e-11, Geno test: 5.34e-07, 1.08e-09, Trend test: 8.71e-08, 1.36e-10
Allele test: 3.02e-10, 1.50e-11, Geno test: 2.41e-09, 1.31e-10, Trend test: 3.04e-10, 4.44e-11
Allele test: 1.98e-03, 1.93e-09, Geno test: 6.43e-03, 1.84e-08, Trend test: 1.72e-03, 2.42e-09
(100,)
>>>
>>>
```

3.6.11 Population structure

Parameter `Fst` measures the F_{st} statistic for population structure at each specified locus (variables `Fst`, `Fis` and `Fit`), and a summary statistic for all loci (variables `AvgFst`, `AvgFis` and `AvgFit`). This statistic by default uses all existing subpopulations, but it can also be applied to a subset of subpopulations, or even virtual subpopulations using parameter *subPops*. That is to say, you can measure the genetic difference between males and females using `subPops=[(0,0), (0,1)]` if a sexSplitter is used to define two virtual subpopulations with male and female individuals respectively.

Example 3.42 demonstrate a simulation with two replicates. In the first replicate, three subpopulations evolve separately without migration and become more and more genetically distant. In the second replicate, a low level migration is applied between subpopulations so the population structure is kept at a low level.

Example 3.42: Measure of population structure (Fst)

```
>>> from simuUtil import MigrIslandRates
>>>
>>> simu = simulator(population([5000]*3, loci=[10], infoFields='migrate_to'),
...     randomMating(), rep=2)
>>> simu.evolve(
...     preOps = initByFreq([0.5, 0.5]),
...     ops = [
...         migrator(rate=MigrIslandRates(0.01, 3), reps=1),
...         stat(Fst=range(10), step=40),
...         pyEval("'Fst=%.3f (rep=%d, with migration) ' % (AvgFst, rep)", step=40),
...         pyOutput('\n', reps=-1, step=40)
```

```
...     ],
...     gen = 200
... )
Fst=0.000 (rep=0, with migration) Fst=0.000 (rep=1, with migration)
Fst=0.007 (rep=0, with migration) Fst=0.004 (rep=1, with migration)
Fst=0.013 (rep=0, with migration) Fst=0.004 (rep=1, with migration)
Fst=0.015 (rep=0, with migration) Fst=0.004 (rep=1, with migration)
Fst=0.018 (rep=0, with migration) Fst=0.003 (rep=1, with migration)
(200, 200)
>>>
```

3.6.12 Hardy-Weinberg equilibrium test

Parameter `HWE` accepts a list of loci at which exact Hardy Weinberg equilibrium tests are applied. The p-values of the tests are assigned to a dictionary `HWE`. Example 3.43 demonstrates how Hardy Weinberg equilibrium is reached in one generation.

Example 3.43: Hardy Weinberg Equilibrium test

```
>>> simu = simulator(population([1000], loci=1), randomMating())
>>> simu.evolve(
...     preOps = initByValue([[0,0], [0, 1], [1,1]], proportions=[0.4, 0.4, 0.2]),
...     ops = [
...         stat(HWE=0, genoFreq=0, stage=PrePostMating),
...         pyEval(r'"HWE p-value: %.5f (AA: %.2f, Aa: %.2f, aa: %.2f)\n" % (HWE[0], '
...             'genoFreq[0][(0,0)], genoFreq[0][(0,1)] + genoFreq[0][(1,0)], genoFreq[0][(1,1)])',
...             stage=PrePostMating),
...     ],
...     gen = 1
... )
HWE p-value: 0.00000 (AA: 0.40, Aa: 0.39, aa: 0.21)
HWE p-value: 0.85232 (AA: 0.34, Aa: 0.49, aa: 0.17)
(1,)
>>>
>>>
```

3.7 Miscellaneous operators

3.7.1 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 3.7.1 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
)
```

3.7.2 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 3.44) presents a rather complicated usage of this operator.

Example 3.44: 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 (Male), 5 individuals:
   0: MU  51 56 58 |  56 53 55 |  2
   2: MU  53 55 53 |  53 50 55 |  2
   3: MU  58 59 54 |  52 58 55 |  2
   6: MU  53 50 58 |  57 55 59 |  2
   9: MU  55 51 59 |  57 59 51 |  2
Subpopulation 1,1 (Female), 4 individuals:
  10: FU  54 58 51 |  59 53 50 |  2
  11: FU  50 50 50 |  56 51 52 |  2
  15: FU  53 55 57 |  59 57 56 |  2
  19: FU  57 53 53 |  56 59 51 |  2

Ancestry population 1
Subpopulation 0,0 (Male), 6 individuals:
   0: MU  22 23 26 |  26 29 26 |  1
   1: MU  26 26 21 |  24 26 26 |  1
   4: MU  27 22 23 |  28 27 23 |  1
   7: MU  25 21 29 |  25 29 22 |  1
   8: MU  24 23 25 |  27 22 21 |  1
   9: MU  24 25 24 |  20 28 27 |  1
Subpopulation 1,1 (Female), 2 individuals:
  11: FU  28 29 25 |  29 28 28 |  1
  18: FU  28 23 27 |  29 20 22 |  1

>>>
>>>
```

3.7.3 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 3.45 to save population to files such as `snapshot_5_20.pop` during the evolution.

Example 3.45: 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)
>>>
```

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

Example 2.67 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 3.46 demonstrates a tpical usage of this operator.

Example 3.46: 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')
53.0
>>>
```

3.7.5 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 3.50 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 3.47: 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('len(alleleFreq[0]) == 1',
...             pyExec('stoppedAt = gen')),
...         terminateIf('len(alleleFreq[0]) == 1')
...     ]
... )
(1189, 2422, 2069, 2384)
>>> for pop in simu.populations():
...     print 'Overall: %4d, below 40%%: %4d, above 60%%: %4d' % \
...         (pop.dvars().stoppedAt, pop.dvars().below40, pop.dvars().above60)
...
Overall: 1188, below 40%: 1006, above 60%:    0
Overall: 2421, below 40%: 1181, above 60%:  502
Overall: 2068, below 40%: 1058, above 60%:  145
Overall: 2383, below 40%:  484, above 60%: 1409
>>>
```

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

3.7.6 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 3.48 demonstrates how to turn on debug information conditionally and turn it off afterwards, using operators `turnOnDebug` and `turnOffDebug`.

Example 3.48: 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',
...             ifOps = [
...                 turnOnDebug(DBG_MUTATOR),
...                 pointMutator(loci=0, allele=0, inds=0),
...             ],
...             elseOps = turnOffDebug(DBG_MUTATOR)),
...     ],
...     gen = 100
... )
Mutate locus 0 to allele  at generation 15
Mutate locus 0 to allele  at generation 18
Mutate locus 0 to allele  at generation 18
```

3.7.7 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 3.49.

Example 3.49: 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), reps=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.

3.7.8 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 3.50.

Example 3.50: 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, reps=-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)
>>>
```

3.8 Selection (under revision)

3.8.1 How natural selection works in simuPOP

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

- Before mating happens, an operator (called a **selector**) goes through a populations and assign each individual a fitness value. The fitness values are stored in an information field called `fitness`.
- When mating happens, parents are chosen with probabilities that are proportional to their fitness values. For example, assuming that a parental population consists of four individuals with fitness values 1, 2, 3, and 4, respectively, the probability that they are picked to produce offspring are $1/(1+2+3+4) = 0.1$, 0.2, 0.3, and 0.4 respectively. As you can image, if the offspring population has 10 individuals, the four parents will on average parent 1, 2, 3 and 4 offspring.

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

Although the underlying mechanisms are the same, more complicated selection schemes could be simulated in simuPOP. For example

- Individuals are chosen in their own groups. For example, in sexual random mating, relative fitness values of fathers and mothers are calculated separately.
- In the diploid case, the same allele can be protective in heterozygotes (*Aa*) and detrimental in homozygotes (*aa*) so a mutant (*a*) could be maintained in a population in the form of heterozygotes.
- Individual fitness could be determined by multiple disease predisposing loci and their interacting with environmental factors.
- Although individual fitness is usually determined by individual genotype, it can be caused purely by environmental factors. For example, if geographical distributions of individuals are simulated, individuals located in the north may have lower fitness values and produce less offspring than individual located in the south.
- Not all mating schemes support natural selection.

3.8.2 Map selector

Example 3.51: 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
... )
0.7715
0.3010
0.3665
(300,)
>>>
```

3.8.3 Multi-allele selector (operator `maOperator`)

A multi-allele selector divides alleles into two groups, wildtype *A* and mutants *a*, and treat alleles within each group as the same. The fitness model is therefore simplified to three fitness values: fitness for genotype *AA*, *Aa* and *aa*. The most widely used model treat allele 0 as wildtype and any other alleles as mutants.

Example 3.52: 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)
0.7600
0.3590
0.3100
(300,)
>>>
```

3.8.4 Multi-loci selector

Example 3.53: 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,)
>>>
```

3.8.5 A hybrid selector

Example 3.54: 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
... )
0.8150
0.9840
0.9965
0.9980
(100,)
>>>
```

3.8.6 Should we select parents or offspring? **

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 ($s_1 = .1$, $s_2 = .2$).

3.9 Penetrance (under revision)

3.9.1 Map penetrance model

Example 3.55: 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)
>>>
```

3.9.2 Multi-loci penetrance model

Example 3.56: 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
>>>
```

3.9.3 Hybrid penetrance model

Example 3.57: 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[l*2]+geno[l*2+1])*0.3
...     return p
...
>>> PyPenetrance(pop, func=peneFunc, loci=(0, 1, 2))
>>> Stat(pop, numOfAffected=True)
>>> print pop.dvars().numOfAffected
79
>>> #
>>> # 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

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.

Function `simuOpt.saveConfig(filename)` saves current values of parameters to a configuraiton file `filename`. Parameters that do not have a label are ignored. This configuration file can be loaded later using command line option `--config filename`, perhaps with option `--gui=False` to run the script in batch mode. A less noticed feature of this function is that it also writes a complete command that specifies the same parameters using command line options. This can be handy if you would like to use real parameter definitions instead of `--config filename` in a batch file.

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

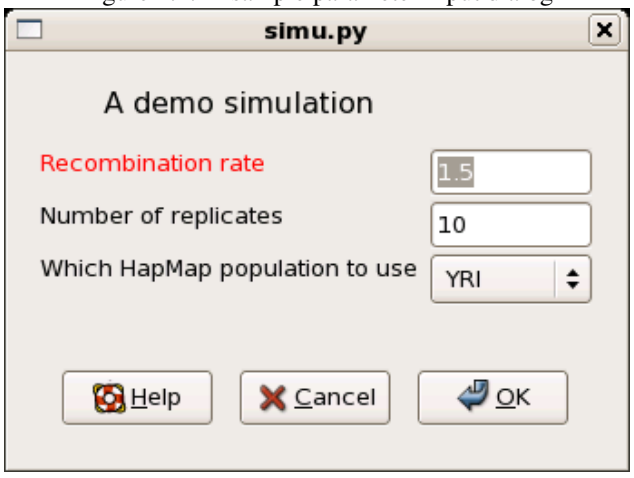
```
>>> pars = simuOpt.simuOpt(options, 'A demo simulation')
>>> print pars.usage()
A demo simulation

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. **Note:** If `simuOpt.valueValidFile()` or `simuOpt.valueValidDir()` is used to validate a parameter, **double click** the input textbox of this parameter will open a file or directory browse dialog.

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`.

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()`.

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']
>>> # Default value of parameter rep is changed
>>> # additional attribute is added.
>>> par1 = simuOpt.simuOpt(options, # all parameters with default values
...     rep=50,                     # default value of rep is changed
...     additional=10               # derived parameters are added
... )
>>> # print all parameters except for derived ones.
>>> print par1.asDict()
{'rate': [0.01], 'rep': 50, 'pop': 'CEU'}
>>> # All parameters are derived ...
>>> par2 = simuOpt.simuOpt(rep=50, pop='CEU', rate=[0.5])
>>> print par2.asDict()
{}
>>> print par2.rep, par2.pop
50 CEU
>>>
```

It is easy to set **additional attributes** to a `simuOpt` object, using either `par.name = value` statement or additional `name=value` pairs in the constructor of a `simuOpt` object. These attributes are not considered as parameters of an `simuOpt` object (e.g. they are not returned by function `simuOpt.asDict()`) but could be used just like regular parameters. Note that the same operations for `simuOpt` parameters change the value of these parameters.

Additional attributes can be used to create a `simuOpt` object without user interaction. For example, objects `par1` and `par2` in Example 4.3 are created easily with needed attributes. They can be passed to functions where a `simuOpt` object is needed, although some of the attributes are not real parameters (in the sense that they are not created by a parameter specification dictionary and will not be used to handle user input).

4.2 simuUtil (under revision)

The simuUtil module provides a few utility functions and classes. They do not belong to the simuPOP core but are distributed with simuPOP because they are frequently used and play an important role in some specialized simulation techniques. Please refer to the simuPOP online cookbook (`http://simupop.sourceforge.net/cookbook`) for more utility functions.

4.2.1 Trajectory simulation

A forward-time simulation, by its nature, is directly influenced by random genetic drift. Starting from the same parental generation, allele frequencies in the offspring generation would vary from simulation to simulation, with perhaps a predictable mean frequency which is determined by factors such as parental allele frequency, natural selection, mutation and migration.

Genetic drift is unavoidable and is in many cases the target of theoretical and simulation studies. However, in certain types of studies, there is often a need to control the frequencies of certain alleles in the present generation. For example, if we are studying a particular penetrance model with pre-specified frequencies of disease predisposing alleles, the simulated populations would better have consistent allele frequencies at the disease predisposing loci, and consequently consistent disease prevalence.

simuPOP provides a special offspring generator `controlledOffspringGenerator` and an associated mating scheme called `controlledRandomMating` that can be used to generate offspring generations conditioning on frequencies of one or more alleles. This offspring generator essentially uses a reject-sampling algorithm to select (or

reject) offspring according to their genotypes at specified loci. A detailed description of this algorithm is given in Peng et al. [2007].

The controlled random mating scheme accepts a user-defined trajectory function that tells the mating scheme the desired allele frequencies at each generation. Example 2.54 uses a manually defined function that raises the frequency of an allele steadily. However, given known demographic and genetic factors, **a trajectory should be simulated randomly so that it represents a random sample from all possible trajectories that match the allele frequency requirement**. If such a condition is met, the controlled evolutionary process can be considered as a random processes condition on allele frequencies at the present generation. Please refer to Peng et al. [2007] for a detailed discussion about the theoretical requirements of a valid trajectory simulator.

The `simuUtil` module provides functions and classes that implement two trajectory simulation methods that can be used in different situations.

Forward-time trajectory simulations.

A forward simulation starts at a specified generation with specified allele frequencies and simulates allele frequencies generation by generation until it reaches the present generation. The simulation will restart if the present allele frequencies do not fall into a specified range. The length of the trajectory (in generations) is fixed. This method can be used to simulate trajectories of existing alleles.

Backward-time trajectory simulations.

A backward simulation starts from specified frequencies at the present generation. In the single-allele case, the simulations goes backward-in-time until an allele gets lost. The length of such a trajectory is random, which is usually a desired property because the age of a mutant in the present generation is usually unknown and is assumed to be random.

se of how to generate a forward-time evolutionary process with complex diseases, which are introduced by point mutations.

1. Given current disease allele frequencies and selection and demographic models, simulate trajectories of allele frequencies at each DSL using a backward approach (BackwardTrajectory.traj) [Peng2008, Slatkin2001].
2. Create an initial population and initialize individuals randomly with several initial haplotypes. Burn-in the population subject to mutation (non-DSL markers only) and recombination, which will be present during the whole evolutionary process.
3. Introduce the disease alleles to the population by point-mutating disease loci of different individuals (BackwardTrajectory.mutators()). The generations when mutants are introduced are determined by allele frequency trajectories.
4. Evolve the population according to the simulated allele frequency trajectories and predetermined demographic features (controlledRandomMating(...)).

The advantage of this approach is that we control the disease allele frequencies during evolution while allowing randomintroduction of disease mutants.

Example 4.2.1 forward-time simulation with complex diseases

4.2.2 Progress bar

4.3 simuRPy

The `simuRPy` module defines a few utility functions and Python operators that help you plot variables and information fields during evolution. A number of operators are defined that

- Operator `simuRPy.varPlotter`: Plot a dynamically evaluated expression with its history. Each expression and its history form a line in the plot. Multiple lines will be plotted for multiple replicates and/or for each element of the expression (if the evaluated value of the expression is a sequence), with options to separate lines to different subplots.
- Operator `simuRPy.scatterPlotter`: Plot individuals in specified (virtual) subpopulations using values at two information fields as x and y axes. Individuals belonging to different (virtual) subpopulations will be plotted with different colors and shapes.
- Operator `simuRPy.infoPlotter`: Using a R function such as `hist` and `qqnorm` to plot one or more information fields of individuala in one or more (virtual) subpopulations. Two specialized operators `simuRPy.histPlotter` and `simuRPy.qqPlotter` are provided to plot the histograms and qq plots. Other functions could also be used, and it is even possible to draw a figure completely by your own (with stratified data provided to you by this operator).
- Operator `simuRPy.boxPlotter`: This operator uses R function `boxplot` to plot boxplots of data of one or more information fields of individuals in one ore more (virtual) subpopulations. The whiskers could be grouped by information field or subpopulations.

These operators are derived from class `pyOperator` and call R plot functions when they are applied to a population. For example, operator `simuRPy.varPlotter` collects expression values and use functions `plot` and `lines` to plot the data, with help from other functions such as `par` (device property), `dev.print` (save figure to files) and `legend` (add legend). Some functions are called multiple times for different replicate, subpopulation or information fields.

4.3.1 Derived keyword arguments *

One of the most interesting feature of this module is its use of derived keyword parameters to send arbitrary parameters to the underlying R functions, which usually accept a large number of parameters to customize every aspect of a figure. A **derived keyword argument** is an argument that is prefixed with a function name and/or suffixed by an iterator name. The former specifies to which underlying R function this parameter will be passed to; the latter allows the users to specify a list of values that will be passed, for example, to lines representing different replicates. For example, parameter `par_mar=[1]*4` will pass `mar=[1]*4` to R function `par`, and `lty_rep=[1, 2, 3]` will pass `lty=1`, `lty=2` and `lty=3` to different replicates. A class usually has one or two default functions (such as `plot`, `lines`) to which keyword aguments without function prefix will be sent.

In addition, the values of these keyword arguments could vary during evolution. More specifically, if the value is a string with a leading exclamation mark (`!`), the remaining string will be considered as an expression. This expression will be evaluated against the current population during evolution and the return value will become the value of the parameter at that generation. For example, keyword parameter `main="!'Allele frequency at generation %d' % gen"` will become `main='Allele frequency at generation 10'` at generation 10.

4.3.2 Plot of expressions and their histories

Class `simuRPy.varPlotter` plots the current and historical values of a Python expression (`expr`), which are evalulated (against each population's local namespace) and saved during evolution. The return value of the expression

can be a number or a sequence, but should have the same type and length across all replicates and generations. Histories of each value (or each item in the returned sequence) of each replicate form a line, with generation numbers as its x-axis. Number of lines will be the number of replicates multiplied by dimension of the expression. Although complete histories are usually saved, you can use parameter `win` to save histories only within the last `win` generations.

Except for the first generation where no line could be drawn, a figure will be drawn after this operator is applied to the last specified replicate (parameter `reps` could be used to specify a subset of replicates). For example, although linkage disequilibrium values between the first two loci are evaluated and saved at the end of generations 0, 5, 10, ..., (`step=5`) figures are only drawn at generations 40 and 80 (`update=40`) in Exampe 4.4. This example also demonstrates the use of parameters `saveAs` and `legend`. By given a filename `rpy.png` to parameter `saveAs`, this operator will save figures (named `rpy_40.png` and `rpy_80.png`) after they are drawn.

Example 4.4: Use rpy to plot an expression

```
>>> from simuPOP import *
>>> from simuRPy import varPlotter
>>> pop = population(size=1000, loci=2)
>>> simu = simulator(pop, randomMating(), rep=3)
>>> simu.evolve(
...     preOps = initByValue([1, 2, 2, 1]),
...     ops = [
...         recombinator(rates=0.01),
...         stat(LD=[0, 1]),
...         varPlotter('LD[0][1]', step=5, update=40, saveAs='log/rpy.png',
...             legend=['Replicate %d' % x for x in range(3)],
...             ylab='LD between marker 1 and 2',
...             ylim=[0, 0.25], main='LD decay', lty_rep=[1, 2, 3],
...         ),
...     ],
...     gen=100
... )
(100, 100, 100)
>>>
```

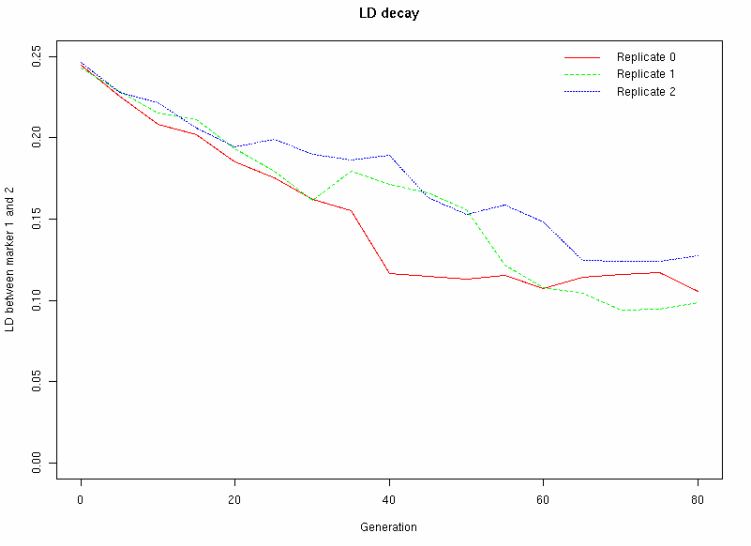
Parameters after `legend` (`xlab`, `ylab`, `ylim`, `main`, ...) deserve more attention here. These parameters are derived keyword arguments because they are not defined by `varPlotter`. Parameters without prefix are passed directly to the R functions `plot` and `line`. They could be used to customize line type (`lty`), color (`col`), title (`main`), limits of x and y axes (`xlim` and `ylim`) and many other graphical features (see R manual for details). If multiple lines are drawn, a list of values could be applied to these lines if you add `_rep` (for each replicate) or `_dim` (for each item of a sequence) after the name of the parameter. For example, `lty_rep=[1, 2, 3]` is used in Example 4.4 to pass parameters `lty=1`, `lty=2` and `lty=3` to lines for three replicates. Suffix `_repdim` can also be used to specify values for every replication and dimension. Figure 4.2 displayed `rpy_80.png` that is saved at generation 80 for this example.

If the expression is multidimensional, the number of lines can be large and it is often desired to separate these lines into subplots. This can be done by parameters `byRep` or `byDim`. The former plots lines replicate by replicate and the latter does it dimension by dimension. For example, Example 4.5 and 4.6 both have three replicates and the expression has allele frequency for four loci. The total number of lines is therefore 12. In Example 4.5, these lines are separated to three subplots, replicate by replicate, with different titles (parameter `main_rep`). In each subplot, allele frequency trajectories (histories) for different loci are plotted in different color (parameter `col_dim`). The last saved figure (`rpy_byRep_90.png`) is displayed in Figure 4.3. In Example 4.6, these lines are separated to four subplots, locus by locus, with different titles (parameter `main_dim`). In each subplot, allele frequency trajectories (histories) for different loci are plotted in different color (parameter `col_rep`) and line type (parameter `lty_rep`). The last saved figure (`rpy_byDim_90.png`) is displayed in Figure 4.4.

Example 4.5: Separate figures by replicate

```
>>> from simuPOP import *
>>> from simuRPy import varPlotter
```

Figure 4.2: rpy_80.png saved at generation 80 for Example 4.4




```
>>> pop = population(size=1000, loci=1*4)
>>> simu = simulator(pop, randomMating(), rep=3)
>>> simu.evolve(
...     preOps = [initByFreq([0.1*(x+1), 1-0.1*(x+1)], loci=x) for x in range(4)],
...     ops = [
...         stat(alleleFreq=range(4)),
...         varPlotter('[alleleFreq[x][0] for x in range(4)]', byRep=True,
...             update=10, saveAs='log/rpy_byRep.png',
...             legend=['Locus %d' % x for x in range(4)],
...             ylab='Allele frequency',
...             ylim=[0, 1],
...             main_rep=['Genetic drift, replicate %d' % x for x in range(3)],
...         ),
...     ],
...     gen=100
... )
(100, 100, 100)
>>>
```

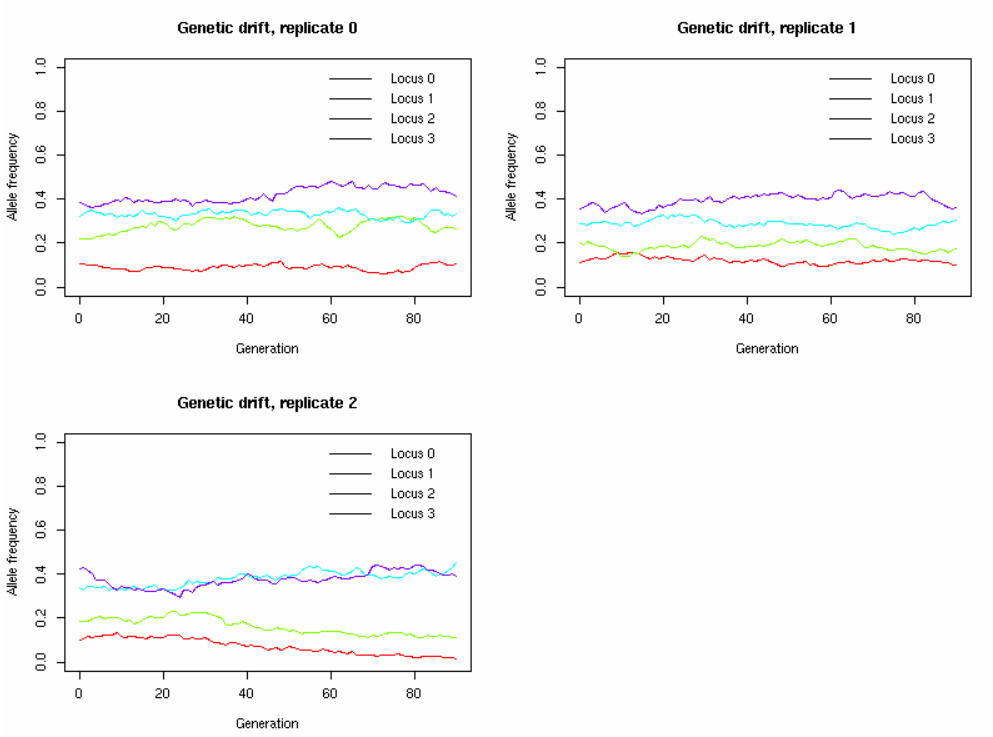
Example 4.6 also demonstrates some advanced features of this plotter that allow further customization of the figures. More specifically,

- Function-specific parameters can be passed to the underlying R function by prefixing function names to parameter names. For example, `plot_axis=False` is used to pass `axis=False` to the `r.plot` function (and not to function `lines` which does not accept this parameter).
- Several hook function can be defined and passed to parameters `preHook`, `postHook` and `plotHook`, which will be called, respectively, before a figure is drawn, after a figure is drawn, and after each `r.plot` call. Example 4.6 uses a `plotHook` function to draw axes of the plots and call `mtext` to add texts to the margins.

Example 4.6: Separate figures by Dimension

```
>>> from simuPOP import *
>>> from simuRPy import varPlotter
>>> pop = population(size=1000, loci=1*4)
>>> simu = simulator(pop, randomMating(), rep=3)
```

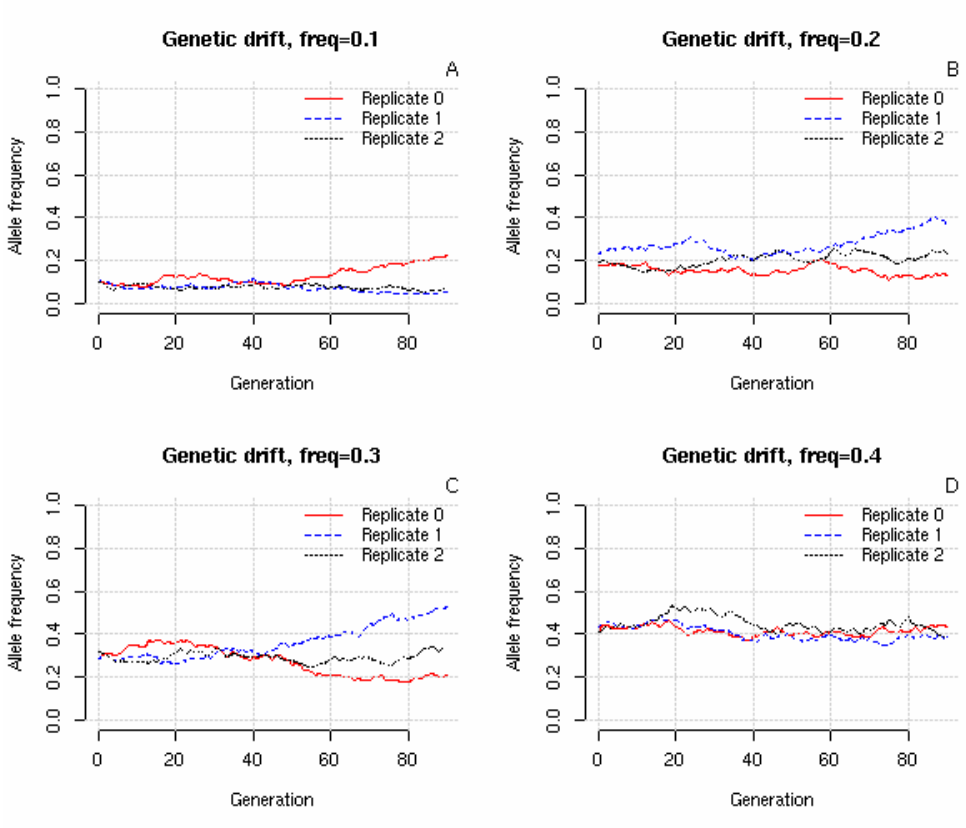
Figure 4.3: Allele frequency trajectories separated by replicates




```
>>>
>>> def drawFrame(r, dim=None, **kwargs):
...     '''Draw a frame around subplot dim. Parameter r is defined in the rpy
...     module and is used for calling R functions. Parameter dim is the dimension
...     index. Other parameters are ignored.
...     '''
...     r.axis(1)
...     r.axis(2)
...     r.grid()
...     r.mtext({0:'A', 1:'B', 2:'C', 3:'D'}[dim], adj=1)
...
>>> simu.evolve(
...     preOps = [initByFreq([0.1*(x+1), 1-0.1*(x+1)], loci=x) for x in range(4)],
...     ops = [
...         stat(alleleFreq=range(4)),
...         varPlotter('[alleleFreq[x][0] for x in range(4)]', byDim=True,
...             update=10, saveAs='log/rpy_byDim.png',
...             legend=['Replicate %d' % x for x in range(3)],
...             ylab='Allele frequency',
...             ylim=[0, 1],
...             main_dim=['Genetic drift, freq=%.1f' % ((x+1)*0.10) for x in range(4)],
...             col_rep=['red', 'blue', 'black'],
...             lty_rep=[1, 2, 3],
...             # the default png dimension is 800x600
...             dev_print_width=600, dev_print_height=500,
...             # do not draw axes in r.plot, leaving the job to drawFrame
...             plot_axes=False,
...             # plot frame, grid etc after each r.plot call
...             plotHook = drawFrame,
```

```
...           ),
...       ],
...       gen=100
... )
(100, 100, 100)
>>>
```

Figure 4.4: Allele frequency trajectories separated by loci



4.3.3 Scatter plots

Operator `simuRPy.scatterPlotter` plots individuals in all or selected (virtual) subpopulations in a 2-D plot, using values at two information fields as their x- and y-axis. In the most simplified form,

```
infoPlotter(infoFields=['x', 'y'])
```

will plot all individuals according their values of information fields `x` and `y`. Additional parameters such as pch, `col`, and `cex` can be used to control the shape, color and size of the points.

What makes this operator useful is its ability to differentiate points (individuals) by (virtual) subpopulations (VSPs). If a list of VSPs are given, points representing individuals from these VSPs will be plotted with different colors and shapes. Because simulations that keep track of multiple information fields are usually complicated, let us simulate something interesting and examine Example 4.7 in details.

At the beginning of this example, all individuals are scattered randomly with `x` and `y` being their physical locations. We use `anc` to record individual ancestry and assign 0 and 1 each to half of the population. During evolution,

- Offspring ancestry values are the average of their parents.

- Offspring with higher ancestry value tend to move to the right. More specifically, locations of an offspring will be

$$\frac{(x_1+x_2)}{2}+N\left(\frac{a_1+a_2}{2}-0.5,0.1\right),\frac{(y_1+y_2)}{2}+N\left(0,0.1\right)$$

  where (x_1, y_1) and (x_2, y_2) are locations of parents, a_1 and a_2 are ancestry values of the parents, and $N(a, b)$ are a random number with normal distribution.

An `scatterPlotter` is used to plot the physical location of all individuals. Individual ancestries are divided into five regions (0, 0.2, 0.4, 0.6, 0.8, 1) indicated by small to larger points. Male and female individuals are plotted by different symbol. This scripts uses the following techniques:

- Set individual information fields randomly using `setIndInfo`.
- Define virtual subpopulations using a `infoSplitter`.
- Use `pyTagger` to calculate offspring information fields from parental fields.
- Mark individuals in different VSPs using parameters `col_sp` and `cex_sp`.
- Use `plot_axes=False` and `par_mar=[0, 0, 2, 0]` to pass parameters `axes=False` and `mar=[0, 0, 2, 0]` to functions `plot` and `par` respectively.
- Use `stage=PreMating` to plot figures before mating.

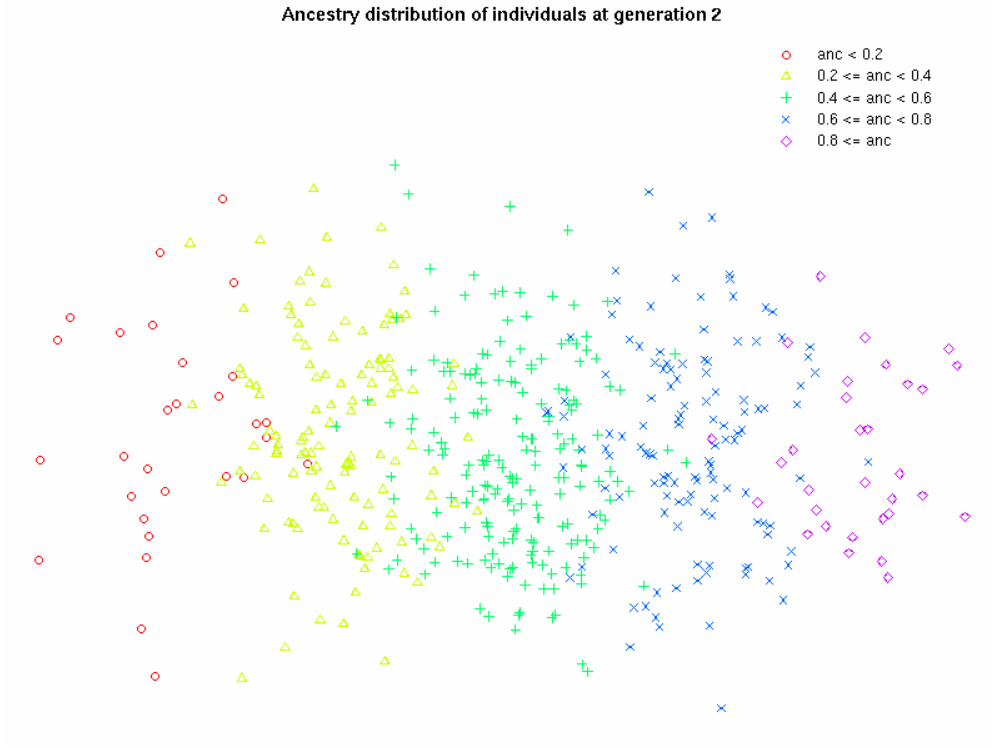
VSPs 0 and 4 appear at the beginning of generation 0, VSP 2 appears at the end of generation 0, and VSP 1 and 3 appear at the end of generation 1. Figure 4.5 displays a figure at the begging of generation 2.

Example 4.7: Use scatterPlotter to plot ancestry of individuals with geographic information.

```
>>> from simuPOP import *
>>> from simuRPy import scatterPlotter
>>> import random
>>> pop = population([500], infoFields=['x', 'y', 'anc'])
>>> # random sex
>>> InitSex(pop)
>>> # random geographic location
>>> pop.setIndInfo([random.random() for i in range(500)], 'x')
>>> pop.setIndInfo([random.random() for i in range(500)], 'y')
>>> # anc is 0 or 1
>>> pop.setIndInfo([random.randint(0, 1) for i in range(500)], 'anc')
>>> # Defines VSP 0, 1, 2, 3, 4 by anc.
>>> pop.setVirtualSplitter(infoSplitter('anc', cutoff=[0.2, 0.4, 0.6, 0.8]))
>>> #
>>> def passInfo(fields):
...     'Parental fields will be passed as x1, y1, anc1, x2, y2, anc2'
...     x1, y1, anc1, x2, y2, anc2 = fields
...     anc = (anc1 + anc2)/2.
...     x = (x1 + x2)/2 + random.normalvariate(anc - 0.5, 0.1)
...     y = (y1 + y2)/2 + random.normalvariate(0, 0.1)
...     return x, y, anc
...
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     ops = [
...         pyTagger(passInfo, infoFields=['x', 'y', 'anc']),
...         scatterPlotter(['x', 'y'], stage=PreMating,
...             saveAs = 'log/scatterPlotter.png',
...             subPops = [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)],
...             ylim = [0, 1.2],
```

```
...             main = "!'Ancestry distribution of individuals at generation %d' % gen",
...             legend = ['anc < 0.2', '0.2 <= anc < 0.4', '0.4 <= anc < 0.6',
...                 '0.6 <= anc < 0.8', '0.8 <= anc'],
...             plot_axes = False,
...             par_mar = [0, 0, 2, 0],
...         ),
...     ],
...     gen = 5,
... )
(5,)
>>>
```

Figure 4.5: Plot of individuals with ancestry marked by different colors



4.3.4 Histograms, QQ plots and boxplots.

Class `simuRPy.infoPlotter` can be used to draw figures for information fields of individuals in one or more subpopulations. `simuRPy.histPlotter` and `simuRPy.qqPlotter` are two special cases of this class that uses functions `hist` and `qqnorm` respectively. Although an `infoPlotter` using a `boxplot` function could be used, a specialized `simuRPy.boxPlotter` is defined so that mutliple boxplot whiskers could be drawn in the same plot.

Using the same evolutionary process as Example 4.7, Example 4.8 uses a `histPlotter` to plot the histograms (Figure 4.6), a `qqPlotter` to plot QQ plot (Figure 4.7), and a `boxPlotter` to plot the boxplots (Figure 4.8) of individual ancestry values. By defining two virtual subpopulations by sex, the `histPlotter` and `qqPlotter` plots two histograms and two QQ plots, one for males and one for females. Different colors are used for these figures. Note that these plots use the special expression value for parameter `main` so that generation number can appear in the titles. The same technique is used in the `dev_print_file` parameter of the `boxPlotter`, which overrides the default filename derived from parameter `saveAs`.

Example 4.8: Use histPlotter to plot the histogram of individual ancestries.

```
>>> from simuPOP import *
>>> from simuRPy import histPlotter, qqPlotter, boxPlotter
>>> import random
>>> pop = population([500], infoFields=['x', 'y', 'anc'])
>>> # random sex
>>> InitSex(pop)
>>> # random geographic location
>>> pop.setIndInfo([random.random() for i in range(500)], 'x')
>>> pop.setIndInfo([random.random() for i in range(500)], 'y')
>>> # anc is 0 or 1
>>> pop.setIndInfo([random.randint(0, 1) for i in range(500)], 'anc')
>>> # Defines VSP 0, 1, 2, 3, 4 by anc.
>>> pop.setVirtualSplitter(sexSplitter())
>>> #
>>> def passInfo(fields):
...     'Parental fields will be passed as x1, y1, anc1, x2, y2, anc2'
...     x1, y1, anc1, x2, y2, anc2 = fields
...     anc = (anc1 + anc2)/2.
...     x = (x1 + x2)/2 + random.normalvariate(anc - 0.5, 0.1)
...     y = (y1 + y2)/2 + random.normalvariate(0, 0.1)
...     return x, y, anc
...
>>> simu = simulator(pop, randomMating())
>>> simu.evolve(
...     ops = [
...         pyTagger(passInfo, infoFields=['x', 'y', 'anc']),
...         histPlotter(infoFields='anc', stage=PreMating,
...             subPops=[(0,0), (0,1)], col_sp=['blue', 'red'],
...             saveAs='log/histPlotter.png',
...             main="!'Histogram of ancestry values at generation %d' % gen",
...         ),
...         qqPlotter(infoFields='anc', stage=PreMating,
...             subPops=[(0,0), (0,1)], col_sp=['blue', 'red'],
...             saveAs='log/qqPlotter.png',
...             main="!'QQ plot of ancestry values at generation %d' % gen",
...         ),
...         boxPlotter(infoFields='anc', stage=PreMating,
...             subPops=[(0,0), (0,1)],
...             saveAs='whatever',
...             dev_print_file='!"log/Gen%d.png" % gen',
...             main="!'Boxplots of ancestry values at generation %d' % gen",
...         ),
...     ],
...     gen = 5,
... )
(5,)
>>>
```

Figure 4.6: Histogram of individual ancestry values

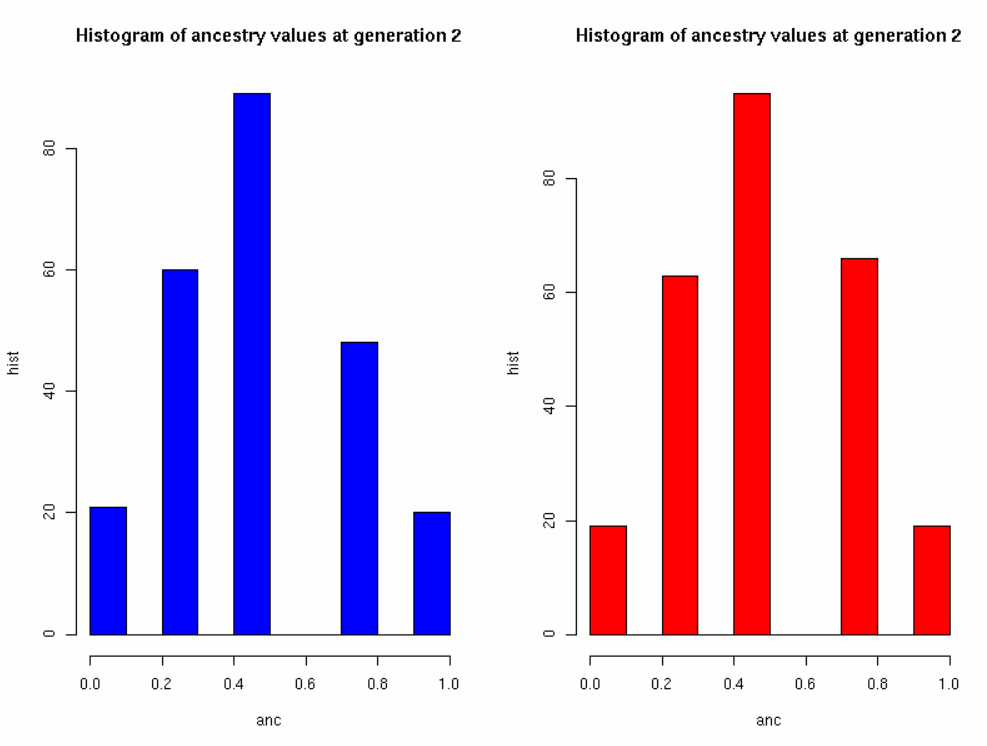


Figure 4.7: QQ plot of individual ancestry values

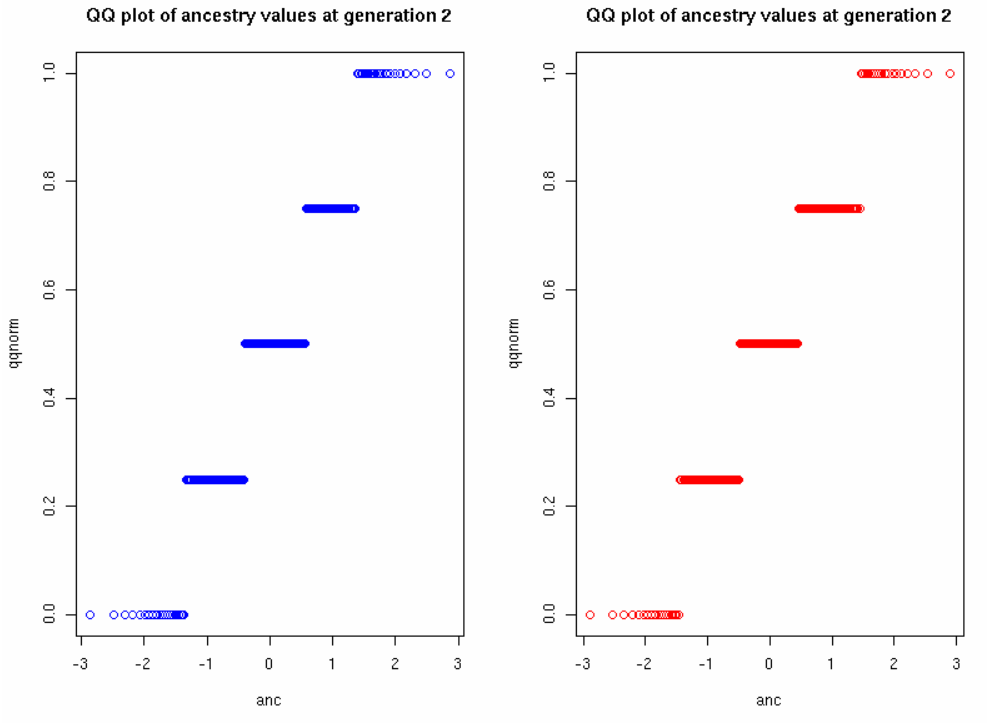
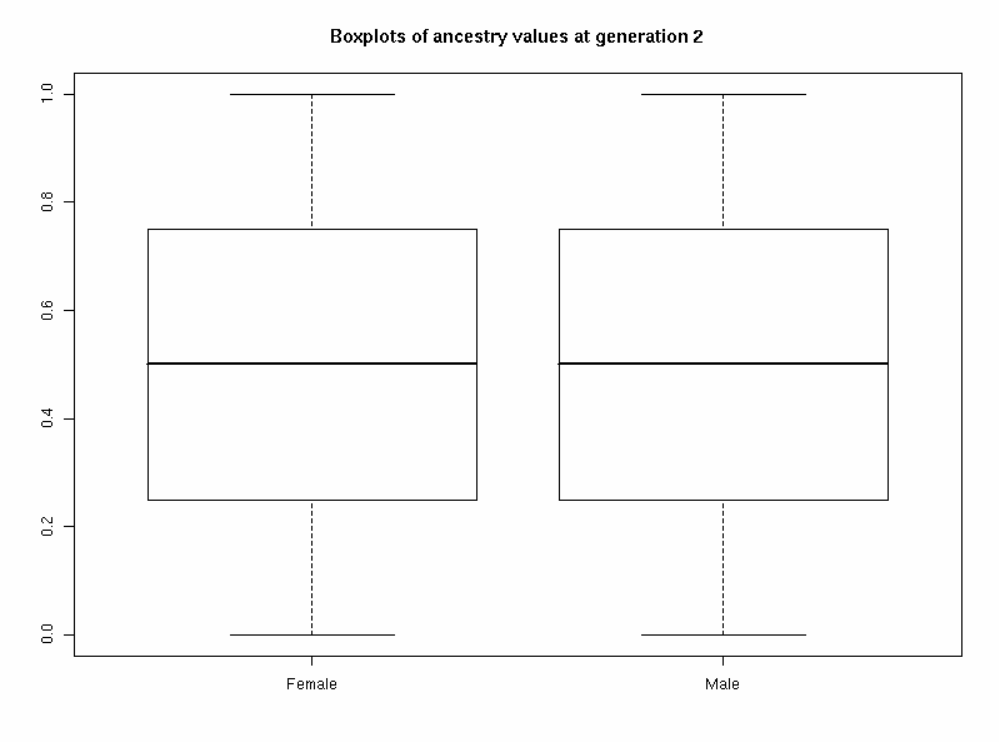


Figure 4.8: Boxplot of individual ancestry values



Chapter 5

A real world example

Previous chapters use a lot of examples to demonstrate individual simuPOP features. However, it might not be clear how to integrate these features in longer scripts that address real world problems, which may involve larger populations, more complex genetic and demographic models and may run thousands of replicates with different parameters. This chapter will show you, step by step, how to write a complete simuPOP script that has been used in a real-world research topic.

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.

This example is a simplified version of the `simuCDCV.py` script that simulates this evolution process and observe the allelic spectra of both types of diseases. The complete script is available at the simuPOP online cookbook. 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

Reich and Lander [2001] used a very simple instant population growth model. Under the model assumption, a population with an initial population size N_0 would evolve G_0 generations, instantly expand its population size to N_1 and evolve another G_1 generations. Such a model can be easily implemented as follows:

```
def ins_expansion(gen, oldsize=[]):
    'An instant population growth model'
    if gen < G0:
        return N0
    else:
        return N1
```

Other demographic models could be implemented similarly. For example, an exponential population growth model that expand the population size from N_0 to N_1 in G_1 generations could be defined as

```
def exp_expansion(gen, oldsize=[]):
    'An exponential population growth model'
```

```
    if gen < G0:
        return N0
    else:
        rate = (math.log(N1) - math.log(N0))/G1
        return int(N0 * math.exp((gen - G0) * rate))
```

That is to say, we first solve r from $N_1 = N_0 \exp(rG_1)$ and then calculate $N_t = N_0 \exp(rG)$ for a given generation.

There is a problem here: the above definitions treat `N0`, `G0`, `N1` and `G1` as global variables. This is OK for small scripts but is certainly not a good idea for larger scripts especially when different parameters will be used. A better way is to wrap these functions by another function that accept `N0`, `G0`, `N1` and `G1` as parameters. That is demonstrated in Example 5.1 where a function `demo_model` is defined to return either an instant or an exponential population growth demographic function.

Example 5.1: A demographic function producer

```
>>> import math
>>> def demo_model(model, N0=1000, N1=100000, G0=500, G1=500):
...     '''Return a demographic function
...     model: linear or exponential
...     N0:   Initial population size.
...     N1:   Ending population size.
...     G0:   Length of burn-in stage.
...     G1:   Length of population expansion stage.
...     '''
...     def ins_expansion(gen, oldsize=[]):
...         if gen < G0:
...             return N0
...         else:
...             return N1
...     rate = (math.log(N1) - math.log(N0))/G1
...     def exp_expansion(gen, oldsize=[]):
...         if gen < G0:
...             return N0
...         else:
...             return int(N0 * math.exp((gen - G0) * rate))
...     if model == 'instant':
...         return ins_expansion
...     elif model == 'exponential':
...         return exp_expansion
...
>>> # when needed, create a demographic function as follows
>>> demo_func = demo_model('exponential', 1000, 100000, 500, 500)
>>> # population size at generation 700
>>> print demo_func(700)
6309
>>>
```

Note: The defined demographic functions return the total population size (a number) at each generation beacuse no subpopulation is considered. A list of subpopulation sizes should be returned if there are more than one subpopulations.

5.3 Mutation and selection models

The thoretical model empolyees an infinite allele model where there is a single wild type allele and an infinite number of disease alleles. Each mutation would introduce a new disease allele and there is no back mutation (mutation from disease allele to wild type allele).

This mutation model can be mimicked by a k-allele model with resaonably large k. We initialize all alleles to 0 which is the wild type (A) and all other alleles are considered as disease alleles (a). Because an allele in a $k-$allele mutation model can mutate to any other allele with equal probability, $P(A \to a) \gg P(a \to A)$ since there are many more disease alleles than the wild type allele. If we choose a smaller k (e.g. $k = 20$), recurrent and back mutations can on longer be ignored but it would be interesting to simulate such cases because they are more realistic than the infinite allele model in some cases.

A k-allele model can be simulated using the `kamMutator` operator which accepts a mutation rate and a maximum allelic state as parameters.

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

Because there are many possible disease alleles, a multi-allelic selector (`maSelector`) could be used to select against the disease alleles. This operator accept a single or a list of wild type alleles (`[0]` in this case) and treat all other alleles as disease alleles. A penetrance table is needed which specified the fitness of each individual when they have 0, 1 or 2 disease alleles respectively. In this example, we assume a recessive model in which only genotype aa causes genetic disadvantages. If we assume a selection pressure parameter s, the operator to use is

```
maSelector(loci=0, wildtype=0, penetrance=[1, 1, 1-s])
```

Note that the use of this selector requires a population information field `fitness`.

This example uses a single-locus selection model but the complete script allows the use of different kinds of multi-locus selection model. If we assume a multiplicative multi-locus selection model where fitness values at different loci are combined (multiplied), a multi-locus selection model (`mlSelector`) could be used as follows:

```
mlSelector([
    maSelector(locus=loc1, fitness=[1,1,1-s1], wildtype=[0]),
    maSelector(locus=loc2, fitness=[1,1,1-s2], wildtype=[0])],
    mode=Multiplicative
)
```

These multi-locus model treat disease alleles at different loci more or less independently. If more complex multi-locus models (e.g. models involve gene - gene and/or gene - interaction) are involved, a multi-locus selector that uses a multi-locus penetrance table could be used.

5.4 Output statistics

We first want to output total disease allele frequency of each locus. This is easy because `stat()` operator can calculate allele frequency for us. What we need to do is use a `stat()` operator to calculate allele frequency and get the result from population variable `alleleFreq`. Because allele frequcies add up to one, we can get the total disease allele frequency using the allele frequency of the wild type allele 0 ($\sum_{i=1}^{\infty} f_i = 1 - f_0$). The actual code would look more or less like this:

```
stat(alleleFreq=[0,1]),
pyEval(r'"%.2f" % (1-alleleFreq[0][0])')
```

We are also interested in the effective number of alleles [Reich and Lander, 2001] at a locus. Because simuPOP does not provide an operator or function to calculate this statistic, we will have to calculate it manually. Fortunately, this is not difficult because effective number of alleles can be calculated from existing allele frequencies, using formula

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

where f_i is the allele frequency of disease allele i.

A quick-and-dirty way to output n_e at a locus (e.g. locus 0) can be:

```
pyEval('1./sum([(alleleFreq[0][x]/(1-alleleFreq[0][0]))**2 for x in alleleFreq[0].keys() if x != 0]
```

but this expression looks complicated and does not handle the case when $f_0 = 1$. A more robust method would involve the `stmts` parameter of `pyEval`, which will be evaluated before parameter `expr`:

```
pyEval(stmts='''if alleleFreq[0][0] == 1:
    ne = 0
else:
    freq = [freq[0][x] for x in alleleFreq[0].keys() if x != 0]
    ne = 1./sum([(f/(1-alleleFreq[0][0])**2 for x in freq])
''', expr=r'"%.3f" % ne')
```

However, this piece of code does not look nice with the multi-line string, and the operator is not really reusable (only valid for locus o). It makes sense to define a function to calculate n_e generally:

```
def ne(pop, loci):
    ' calculate effective number of alleles at given loci'
    Stat(pop, alleleFreq=loci)
    ne = {}
    for loc in loci:
        freq = [y for x,y in pop.dvars().alleleFreq[loc].iteritems() if x != 0]
        sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
        if sumFreq == 0:
            ne[loc] = 0
        else:
            ne[loc] = 1. / sum([(x/sumFreq)**2 for x in freq])
    # save the result to the population.
    pop.dvars().ne = ne
    return True
```

When it is needed to calculate effective number of alleles, a Python operator that uses this function can be used. For example, operator

```
pyOperator(func=ne, param=[0], step=5)
pyEval(r'"%.3f" % ne[0]', step=5)
```

would calculate effective number of alleles at locus 0 and output it.

The biggest difference between `pyEval` and `pyOperator` is that `pyOperator` is no longer evaluated in the population's local namespace. You will have to get the variables explicitly using the `pop.dvars()` function, and the results have to be explicitly saved to the population's local namespace.

The final implementation, as a way to demonstrate how to define a new statistics that hides all the details, defines a new operator by inheriting a class from `pyOperator`. The resulting operator could be used as a regular operator (e.g., `ne(loci=[0])`). A function `Ne` is also defined as the function form of this operator. The code is listed in Example 5.2

Example 5.2: A customized operator to calculate effective number of alleles

```
>>> class ne(pyOperator):
...     '''Define an operator that calculates effective number of
...     alleles at given loci. The result is saved in a population
...     variable ne.
...     '''
...     def __init__(self, loci, *args, **kwargs):
...         self.loci = loci
...         pyOperator.__init__(self, func=self.calcNe, *args, **kwargs)
...     #
...     def calcNe(self, pop):
...         Stat(pop, alleleFreq=self.loci)
...         ne = {}
```

```
...         for loc in self.loci:
...             freq = pop.dvars().alleleFreq[loc]
...             sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
...             if sumFreq == 0:
...                 ne[loc] = 0
...             else:
...                 ne[loc] = 1. / sum([(freq[x]/sumFreq)**2 for x in freq.keys() if x != 0])
...         # save the result to the population.
...         pop.dvars().ne = ne
...         return True
...
>>> def Ne(pop, loci):
...     '''Function form of operator ne'''
...     ne(loci).apply(pop)
...     return pop.dvars().ne
...
>>> pop = population(100, loci=[10])
>>> InitByFreq(pop, [.2] * 5)
>>> print Ne(pop, loci=[2, 4])
{2: 3.9468582309157001, 4: 3.8601638123603865}
>>>
```

5.5 Initialize and evolve the population

With appropriate operators to perform mutation, selection and output statistics, it is relatively easy to write a simulator to perform a simulation. This simulator would create a population, initialize alleles with an initial allic spectrum, and then evolve it according to specified demographic model. During the evolution, mutation and selection will be applied, statistics will be calculated and outputed.

Example 5.3: Evolve a population subject to mutation and selection

```
>>>
>>> def simulate(model, N0, N1, G0, G1, spec, s, mu, k):
...     '''Evolve a population using given demographic model
...     and observe the evolution of its allelic spectrum.
...     model: type of demographic model.
...     N0, N1, G0, G1: parameters of demographic model.
...     spec: initial allelic spectrum, should be a list of allele
...         frequencies for each allele.
...     s: selection pressure.
...     mu: mutation rate.
...     k: k for the k-allele model
...     '''
...     demo_func = demo_model(model, N0, N1, G0, G1)
...     simu = simulator(
...         population(size=demo_func(0), loci=1, infoFields='fitness'),
...         randomMating(subPopSize=demo_func)
...     )
...     simu.evolve(
...         preOps = initByFreq(loci=0, alleleFreq=spec),
...         ops=[
...             kamMutator(k=k, rates=mu),
...             maSelector(loci=0, fitness=[1, 1, 1 - s], wildtype=0),
...             ne(loci=[0], step=100),
...             pyEval(r'"%d: %.2f\t%.2f\n" % (gen, 1 - alleleFreq[0][0], ne[0])',
...                 step=100),
...         ],
```

```
...         gen = G0 + G1
...     )
...
>>> simulate('instant', 1000, 10000, 500, 500, [0.9]+[0.02]*5, 0.01, 1e-4, 200)
0: 0.10 4.93
100: 0.06       2.34
200: 0.09       1.82
300: 0.03       1.00
400: 0.00       0.00
500: 0.01       1.35
600: 0.01       4.87
700: 0.05       3.60
800: 0.06       4.44
900: 0.06       3.37
>>>
```

5.6 Option handling

Everything seems to be perfect until you need to

1. Run more simulations with different parameters such as initial population size and mutaion rate. This requires the script to get its parameters from command line (or a configuration file) and executes in batch mode, perhaps on a cluster system.
2. Allow users who are not familiar with the script to run it. This would better be achieved by a graphical user interface.
3. Allow other Python scripts to import your script and run the simulation function directly.

Although a number of Python modules such as `getopt` are available, the simuPOP `simuOpt` module is especially designed to allow a simuPOP script to be run both in batch and in GUI mode, in standard and optimized mode. Example 5.4 makes use of this module.

Example 5.4: A complete simulation script

```
#!/usr/bin/env python
#
# Author:  Bo Peng
# Purpose: A real world example for simuPOP user's guide.
#
'''
Simulation the evolution of allelic spectra (number and frequencies
of alleles at a locus), under the influence of population expansion,
mutation, and natural selection.
'''

import simuOpt
simuOpt.setOptions(quiet=True, alleleType='long')
from simuPOP import *

import sys, types, os, math

options = [
    {'longarg': 'demo=',
     'default': 'instant',
     'label': 'Population growth model',
     'description': 'How does a population grow from N0 to N1.',
```

```
    'chooseOneOf': ['instant', 'exponential'],
    },
    {'longarg': 'N0=',
     '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': 'N1=',
     'default': 100000,
     'label': 'Final population size',
     'allowedTypes': [types.IntType, types.LongType],
     'description': 'Ending population size (after population expansion)',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'G0=',
     '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': 'G1=',
     'default': 1000,
     'label': 'Length of expansion stage',
     'allowedTypes': [types.IntType],
     'description': 'Number of geneartions of the population expansion stage',
     'validate': simuOpt.valueGT(0)
    },
    {'longarg': 'spec=',
     'default': [0.9] + [0.02]*5,
     'label': 'Initial allelic spectrum',
     'allowedTypes': [types.TupleType, types.ListType],
     'description': '''Initial allelic spectrum, should be a list of allele
           frequencies, for allele 0, 1, 2, ... respectively.''',
     'validate': simuOpt.valueListOf(simuOpt.valueBetween(0, 1)),
    },
    {'longarg': 's=',
     'default': 0.01,
     'label': 'Selection pressure',
     'allowedTypes': [types.IntType, types.FloatType],
     'description': '''Selection coefficient for homozygtes (aa) genotype.
           A recessive selection model is used so the fitness values of
           genotypes AA, Aa and aa are 1, 1 and 1-s respectively.''',
     'validate': simuOpt.valueGT(-1),
    },
    {'longarg': 'mu=',
     'default': 1e-4,
     'label': 'Mutation rate',
     'allowedTypes': [types.IntType, types.FloatType],
     'description': 'Mutation rate of a k-allele mutation model',
     'validate': simuOpt.valueBetween(0, 1),
    },
    {'longarg': 'k=',
     'default': 200,
     'label': 'Maximum allelic state',
     'allowedTypes': [types.IntType],
```

```
     'description': 'Maximum allelic state for a k-allele mutation model',
     'validate': simuOpt.valueGT(1),
    },
]

def demo_model(type, N0=1000, N1=100000, G0=500, G1=500):
    '''Return a demographic function
    type: linear or exponential
    N0:   Initial population size.
    N1:   Ending population size.
    G0:   Length of burn-in stage.
    G1:   Length of population expansion stage.
    '''
    def ins_expansion(gen, oldsize=[]):
        if gen < G0:
            return N0
        else:
            return N1
    rate = (math.log(N1) - math.log(N0))/G1
    def exp_expansion(gen, oldsize=[]):
        if gen < G0:
            return N0
        else:
            return int(N0 * math.exp((gen - G0) * rate))
    if type == 'instant':
        return ins_expansion
    elif type == 'exponential':
        return exp_expansion

class ne(pyOperator):
    '''Define an operator that calculates effective number of
    alleles at given loci. The result is saved in a population
    variable ne.
    '''
    def __init__(self, loci, *args, **kwargs):
        self.loci = loci
        pyOperator.__init__(self, func=self.calcNe, *args, **kwargs)
    #
    def calcNe(self, pop):
        Stat(pop, alleleFreq=self.loci)
        ne = {}
        for loc in self.loci:
            freq = pop.dvars().alleleFreq[loc][1:]
            sumFreq = 1 - pop.dvars().alleleFreq[loc][0]
            if sumFreq == 0:
                ne[loc] = 0
            else:
                ne[loc] = 1. / sum([(x/sumFreq)**2 for x in freq])
        # save the result to the population.
        pop.dvars().ne = ne
        return True

def simuCDCV(model, N0, N1, G0, G1, spec, s, mu, k):
    '''Evolve a population using given demographic model
    and observe the evolution of its allelic spectrum.
    model: type of demographic model.
    N0, N1, G0, G1: parameters of demographic model.
    spec: initial allelic spectrum, should be a list of allele
        frequencies for each allele.
```

```
    s: selection pressure.
    mu: mutation rate.
    k: maximum allele for the k-allele model
    '''
    demo_func = demo_model(model, N0, N1, G0, G1)
    print demo_func(0)
    simu = simulator(
        population(size=demo_func(0), loci=1, infoFields='fitness'),
        randomMating(subPopSize=demo_func)
    )
    simu.evolve(
        preOps=initByFreq(loci=0, alleleFreq=spec),
        ops=[
            kamMutator(rate=mu, maxAllele=k),
            maSelector(loci=0, fitness=[1, 1, 1 - s], wildtype=0),
            ne(loci=0, step=100),
            pyEval(r'"%d: %.2f\t%.2f\n" % (gen, 1 - alleleFreq[0][0], ne[0])',
                step=100),
        ],
        gen = G0 + G1
    )
    return simu.extract(0)

if __name__ == '__main__':
    # get parameters
    par = simuOpt.simuOpt(options, __doc__)
    if not par.getParam():
        sys.exit(1)
    #
    if not sum(par.spec) == 1:
        print 'Initial allelic spectrum should add up to 1.'
        sys.exit(1)
    # save user input to a configuration file
    par.saveConfig('simuCDCV.cfg')
    #
    simuCDCV(*par.asList())
```

Example 5.4 uses a programming style that is used by almost all simuPOP scripts. I highly recommend this style because it makes your script seld-documentary and work well under a variety of environments. A script written in this style follows the following order:

1. First comment block

   The first line of the script should always be

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

   This line tells a Unix shell which program should be used to process this script if the script to set to be executable. This line is ignored under windows. It is customary to put author and date information at the top of a script as Python comments.

2. Module doc string

   The first string in a script is the module docstring, which can be referred by variable `__doc__` in the script. It is a good idea to describe what this script does in detail here. As you will see later, this docstring will be used in the `simuOpt.getParam()` function and be outputed in the usage information of the script.

3. Loading simuPOP and other Python modules

   simuPOP and other modules are usually imported after module docstring. This is where you specify which simuPOP module to use. Although a number of parameters could be used, usually only `alleleType` is

specified because other parameters such as `gui` and `optimized` should better be controlled from command line.

4. Parameter description list

   A list of parameter description dictionaries are given here. This list specifies what parameters will be used in this script and describes the type, default value, name of command line option, label of the parameter in the parameter input dialog in detail. Although some directionary items can be ignored, it is a good practice to give detailed information about each parameter here.

5. Helper functions and classes

   Helper functions and classes are given before the main simulation function.

6. Main simulation function

   The main simulation function preforms the main functionality of the whole script. It is written as a function so that it can be imported and executed by another script. The parameter processing part of the script would be ignored in this case.

7. Script execution part conditioned by `__name__ == '__main__'`

   The execution part of a script should always be inside of a `if __name__ == '__main__'` block so that the script will not be executed when it is imported by another script. The first few lines of this execution block are almost always

   ```
   par = simuOpt.simuOpt(options, __doc__)
   if not par.getParam():
       sys.exit(1)
   ```

   which creates a simuOpt object and tries to get parameters from command line option, a configuration file, a parameter input dialog or interactive user input, depending on how this script is executed. Optionally, you can use

   ```
   par.saveConfig('file.cfg')
   ```

   to save the current configuration to a file so that the same parameters could be retrieved later using parameter `--config file.cfg`.

   After simply parameter validation, the main simulation function can be called. This example uses `simuCDCV(*par.asList())` because the parameter list in the `par` object match the parameter list of function `simuCDCV` exactly. If there are a large number of parameters, it may be better to pass the `simuOpt` object directly in the main simulation function.

The script written in this style could be executed in a number of ways.

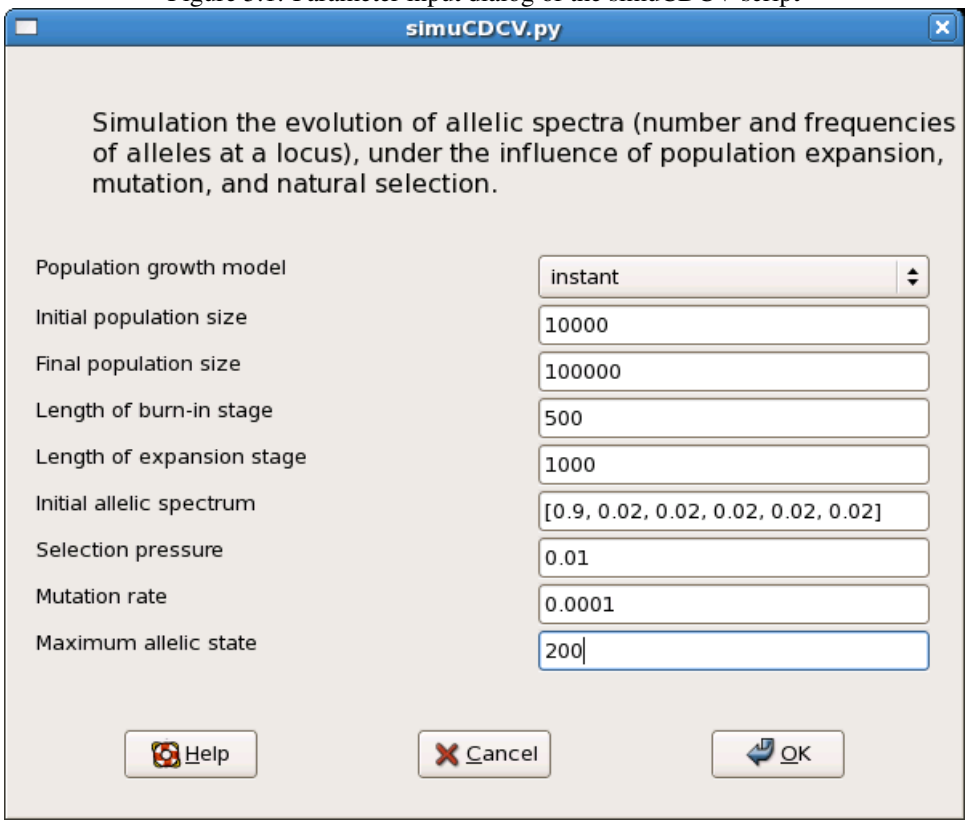
1. If a user executes the script directly, a Tkinter or wxPython dialog will be displayed for users to input parameters. This parameter is shown in Figure 5.1.

2. The help message of this script could be displayed using the Help button of the parameter input dialog, or using command `simuCDCV.py -h`. The help message is displayed in Example 5.5.

Example 5.5: Help information for the simuCDCV script

```
Simulation the evolution of allelic spectra (number and frequencies
of alleles at a locus), under the influence of population expansion,
mutation, and natural selection.


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

options:
```

Figure 5.1: Parameter input dialog of the simuCDCV script




```
-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
--demo ARG            Population growth model [default: instant ]
                      How does a population grow from N0 to N1.
--N0 ARG              Initial population size [default: 10000 ]
                      Initial population size. This size will be maintained
                      till the end of burnin stage
--N1 ARG              Final population size [default: 100000 ]
                      Ending population size (after population expansion)
--G0 ARG              Length of burn-in stage [default: 500 ]
                      Number of generations of the burn in stage.
--G1 ARG              Length of expansion stage [default: 1000 ]
                      Number of geneartions of the population expansion stage
--spec ARG            Initial allelic spectrum [default: [0.9, 0.02, 0.02, 0.02, 0.02, 0.02]
                      Initial allelic spectrum, should be a list of allele
                      frequencies, for allele 0, 1, 2, ... respectively.
--s ARG               Selection pressure [default: 0.01 ]
                      Selection coefficient for homozygtes (aa) genotype. A
                      recessive selection model is used so the fitness values
                      of genotypes AA, Aa and aa are 1, 1 and 1-s
                      respectively.
--mu ARG              Mutation rate [default: 0.0001 ]
                      Mutation rate of a k-allele mutation model
--k ARG               Maximum allelic state [default: 200 ]
                      Maximum allelic state for a k-allele mutation model
```

3. Using parameter `--gui=False`, the script will be run in batch mode. You can specify parameters using

```
simuCDCV.py --gui=False --config file.cfg
```

   if a parameter file is available, or use command line options such as

```
simuCDCV.py --gui=False --demo='instant' --N0=10000 --N1=100000 \
    --G0=500 --G1=500 --spec='[0.9]+[0.02]*5' --s=0.01 \
    --mu='1e-4' --k=200
```

   Note that parameters with `useDefault` set to `True` can be ignored if the default parameter is used. In addition, parameter `--optimized` could be used to load the optimized version of a simuPOP module. For this particular configuration, the optimized module is 30% faster (62s vs. 40s) than the standard module.

4. The simulation function could be imported to another script as follows

```
from simuCDCV import simuCDCV
simuCDCV(model='instant', N0=10000, N1=10000, G0=500, G1=500,
    spec=[0.9]+[0.02]*5, s=0.01, mu=1e-4, k=200)
```

Bibliography

B Devlin and N Risch. A comparison of linkage disequilibrium measures for fine-scale mapping. *Genomics*, 29: 311–322, 1995. 103

T Ohta and M Kimura. The model of mutation appropriate to estimate the number of electrophoretically detectable alleles in a genetic population. *Genet Res*, 22:201–204, 1973. 89

Bo Peng and Marek Kimmel. Simulations provide support for the common disease common variant hypothesis. *Genetics*, 175:1–14, 2007. 133

Bo Peng, Chris I Amos, and Marek Kimmel. Forward-time simulations of human populations with complex diseases. *PLoS Genetics*, 3:e47, 2007. 54, 122

David E Reich and Eric S Lander. On the allelic spectrum of human disease. *Trends Genet*, 17(9):502–510, 2001. 133, 135

Neil Risch. Linkage strategies for genetically complex traits. i. multilocus models. *Am J Hum Genet*, 46:222–228, 1990. 36

Index